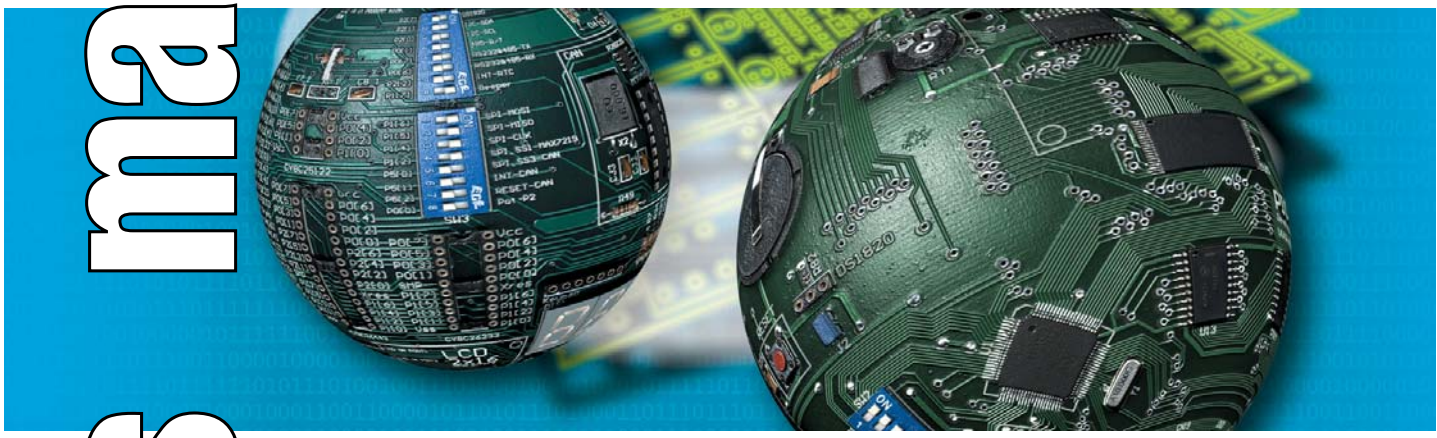
MikroElektronika
Development tools - Books - Compilers
www.mikroelektronika.co.yu

User's manual

PASCAL COMPILER FOR MICROCHIP PIC MICROCONTROLLERS

mikroPASCAL

making it simple...



Develop your applications quickly and easily with the world's most intuitive Pascal compiler for PIC Microcontrollers (families PIC12, PIC16, and PIC18).

Highly sophisticated IDE provides the power you need with the simplicity of a Windows based point-and-click environment.

With useful implemented tools, many practical code examples, broad set of built-in routines, and a comprehensive Help, mikroPascal makes a fast and reliable tool, which can satisfy needs of experienced engineers and beginners alike.

mikroBASIC User's manual

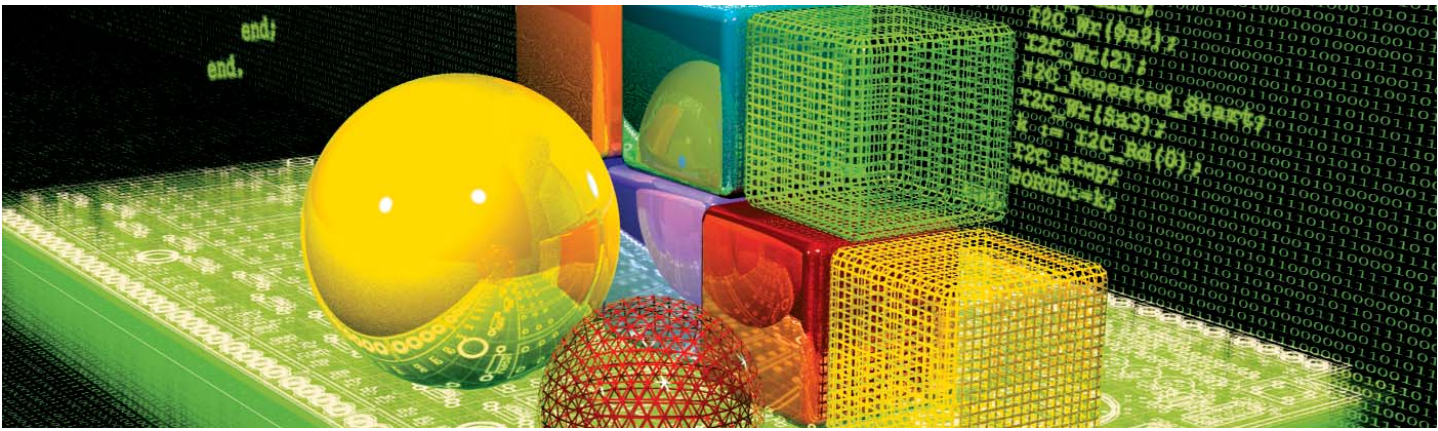


Table of Contents

CHAPTER 1 mikroPascal IDE

CHAPTER 2 mikroPascal Reference

CHAPTER 3 Built-In and Library Routines

CHAPTER 1: mikroPascal IDE 1

Quick Overview 1
Code Editor 3
  Basic Editor Features 3
  Advanced Editor Features 4
Code Explorer 6
Creating First Project 7
Projects 13
  Managing Source Files 14
  Compile Source Code 16
Debugger 17
Error Window 19
Assembly View 20
Statistics 21
Integrated Tools 24
Keyboard Shortcuts 26

CHAPTER 2: mikroPascal Reference 28

Identifiers 29
Keywords 30
Data Types 31
  Array 32
  Strings 34
Numerals and Character Strings 35
Constants 36
Array Constants 38
Variables 40
Comments 44
Expressions 45
Declarations and Statements 47
Directives 50
Procedures and Functions 52

Units 57
Scope (Identifier Visibility) 60
Program Organization 62
Type Conversion 64
Assignment And Implicit Conversion 67
Implicit Conversion And Legal Expressions 71
Operators 75
  Arithmetic Operators 78
  Boolean Operators 81
  Logical (Bitwise) Operators 82
  Relation (Comparison) Operators 86
Conditional Statements 89
  Labels and Goto 89
  Case Statement 90
  If Statement 92
Loops 94
  For Statement 95
  Repeat Statement 97
  While Statement 98
ASM Statement 99
PIC MCU Specific 100
mikroPascal Specific 101
Compiler Error Messages 103

CHAPTER 3: Built-In and Library Routines 106

Built-In Routines 107
Library Routines 109
  1-Wire Library 110
  ADC Library 113
  CAN Library 115
  CANSPI Library 127
  Compact Flash Library 136
  EEPROM Library 144
  I2C Library 145

LCD Library	149
LCD8 Library (8-bit interface)	155
Graphic LCD Library	159
PWM Library	164
RS485 Library	166
SPI Library	172
USART Library	177
Software I2C	180
Software SPI	182
Software UART	184
Flash Memory	186
Manchester Code	189
Numeric Formatting Routines	195
Trigonometry Library	197
Sound Library	199
Utilities	201

To readers note

DISCLAIMER:
mikroPASCAL and this manual are owned by MikroElektronika and are protected by copyright law and international copyright treaty. Therefore, you should treat this manual like any other copyrighted material (e.g., a book). The manual and the compiler may not be copied, partially or as a whole without the written consent from the MikroEelktronika. The PDF-edition of the manual can be printed for private or local use, but not for distribution. Modifying the manual or the compiler is strictly prohibited.

HIGH RISK ACTIVITIES
The mikroPASCAL compiler is not fault-tolerant and is not designed, manufactured or intended for use or resale as on-line control equipment in hazardous environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines, or weapons systems, in which the failure of the Software could lead directly to death, personal injury, or severe physical or environmental damage ("High Risk Activities"). MikroElektronika and its suppliers specifically disclaim any express or implied warranty of fitness for High Risk Activities.

LICENSE AGREEMENT:
By using the mikroPASCAL compiler, you agree to the terms of this agreement. Only one person may use licensed version of mikroPASCAL compiler at a time.
Copyright © MikroElektronika 2003 - 2004.

This manual covers mikroPASCAL version 2.16 and the related topics. New versions may contain changes without prior notice.

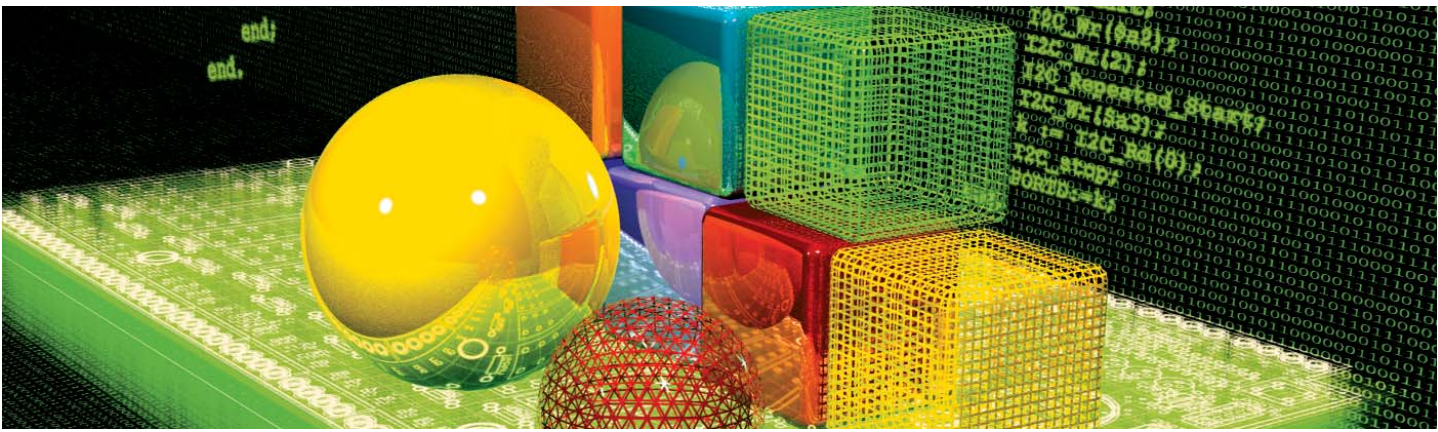
COMPILER BUG REPORTS:
The compiler has been carefully tested and debugged. It is, however, not possible to guarantee a 100 % error free product. If you would like to report a bug, please contact us at the address office@mikroelektronika.co.yu. Please include next information in your bug report:

- Your operating system
- Version of mikroPASCAL
- Code sample
- Description of a bug

CONTACT US:
MikroElektronika magazine
Voice: +381 11 362 04 22, + 381 11 684 919
Fax: +381 11 362 04 22
Web: www.MikroElektronika.co.yu
E-mail: office@MikroElektronika .co.yu

PIC, PICmicro and MPLAB is a Registered trademark of Microchip company. Windows is a Registered trademark of Microsoft Corp. All other trade and/or services marks are the property of the respective owners.

CHAPTER 1

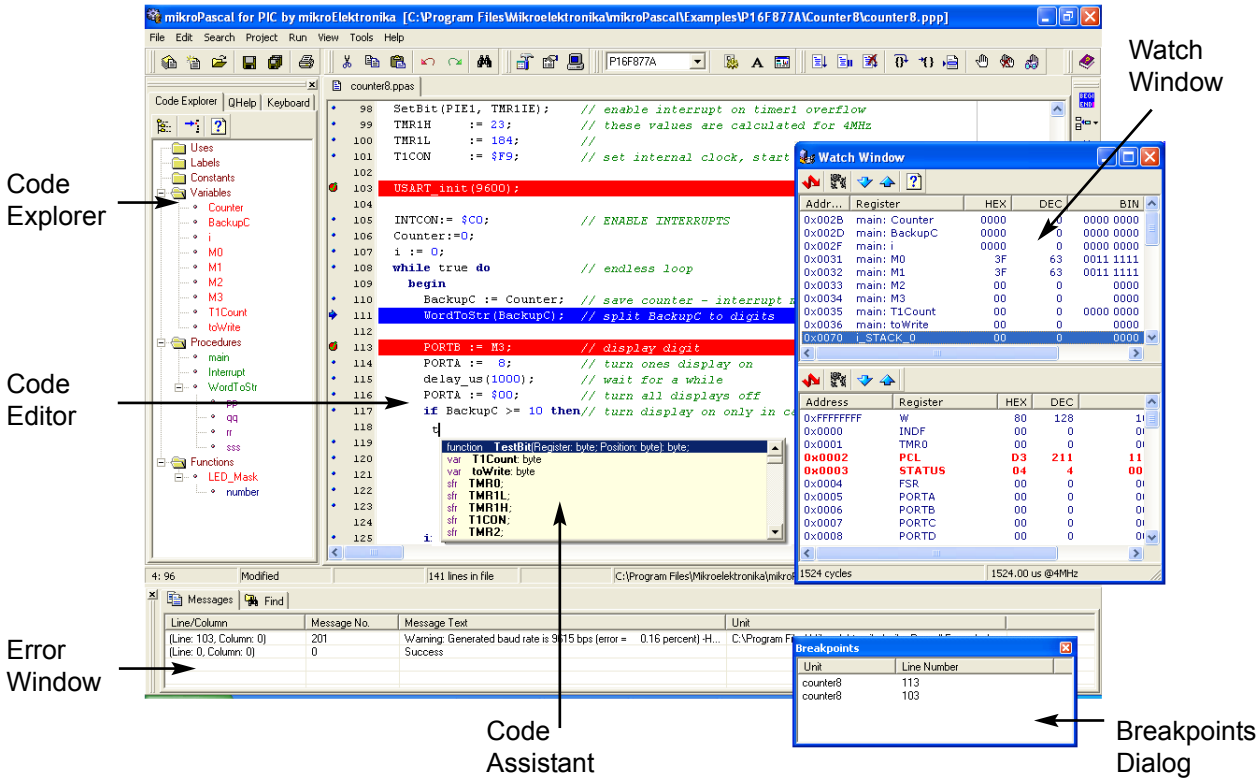


mikroPascal IDE

QUICK OVERVIEW

mikroPascal is a Windows-based Integrated Development Environment, and is much more than just Pascal compiler for PIC MCUs. With mikroPascal, you can:

1. Create Pascal source code using the built-in Code Editor
2. Compile and link your source code
3. Inspect program flow and debug executable logic with Debugger
4. Monitor variables in Watch Window
5. Get error reports
6. Get detailed statistics (how compiled code utilizes PIC MCU memory, hex map, charts and more...)



Code Editor features adjustable Syntax Highlighting, Code Assistant, Parameters Assistant, Auto Correct for common typos, and Code Templates.

Code browser, Keyboard shortcut browser, and Quick Help browser are at your disposal for easier project management.

Error Window displays all errors detected during compiling and linking.

Watch Window enables you to monitor variables, registers and PIC MCU memory.

New Project Wizard is fast, reliable, and easy way to create a project.

Source-level Debugger lets you debug executable logic step-by-step by watching program flow.

Help files are **syntax and context sensitive**.

CODE EDITOR

Basic Editor Features

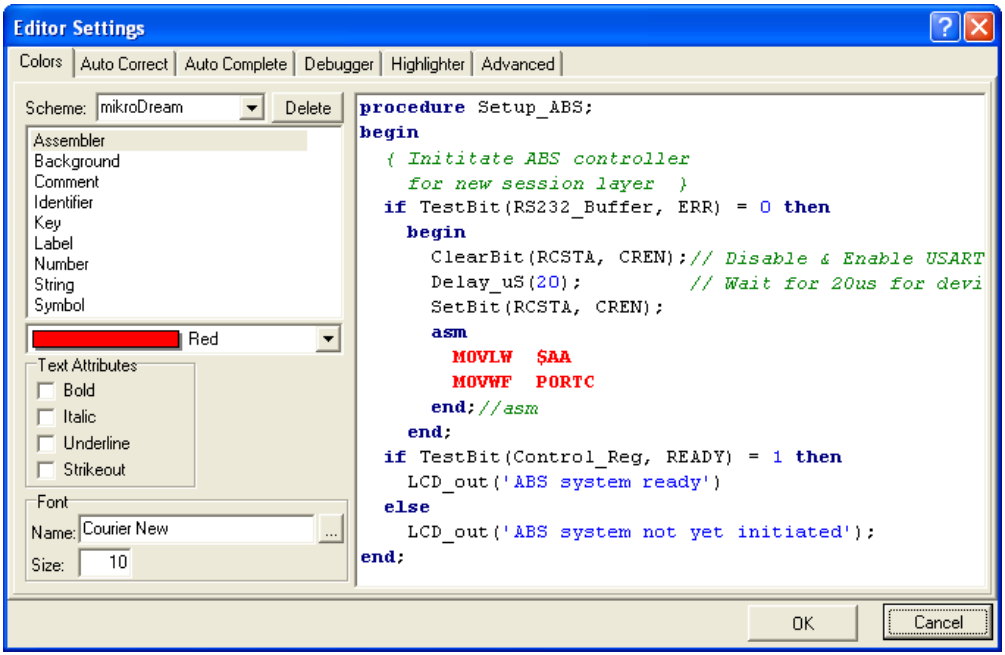
General code editing is same as working with any standard text-editor, including familiar Copy, Paste, and Undo actions, common for Windows environment.

Advanced code editing includes:

- Adjustable Syntax Highlighting
- Code Assistant, Parameters Assistant, Code Templates
- Auto Correct for common typos

You can configure Syntax Highlighting, Code Assistant and Auto Correct from Editor Settings dialog. To access this window, click Tools > Options from drop-down menu, or click Tools icon in Settings toolbar.

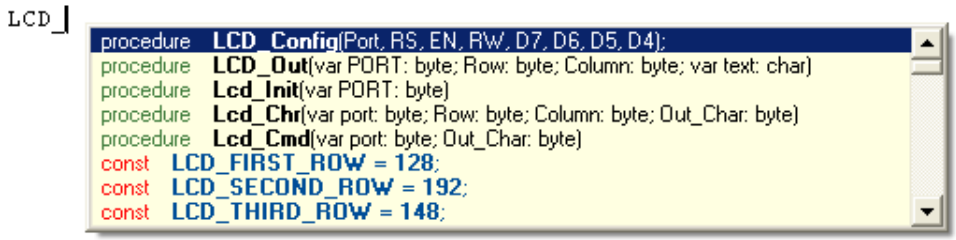
Tools Icon.



Advanced Editor Features

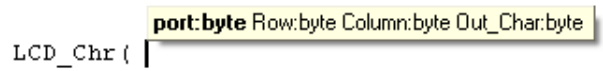
Code Assistant [CTRL+SPACE]

If you type first few letter of a word and then press CTRL+SPACE, all valid identifiers matching the letters you typed will be prompted to you in a floating panel (see the image). Now you can keep typing to narrow the choice, or you can select one from the list using keyboard arrows and Enter.


LCD_
procedure LCD_Config(Port, RS, EN, RW, D7, D6, D5, D4);
procedure LCD_Out(var PORT: byte; Row: byte; Column: byte; var text: char)
procedure Lcd_Init(var PORT: byte)
procedure Lcd_Chr(var port: byte; Row: byte; Column: byte; Out_Char: byte)
procedure Lcd_Cmd(var port: byte; Out_Char: byte)
const LCD_FIRST_ROW = 128;
const LCD_SECOND_ROW = 192;
const LCD_THIRD_ROW = 148;

Parameter Assistant [CTRL+SHIFT+SPACE]

Parameter Assistant will be automatically invoked when you open a parenthesis "(" or press CTRL+SHIFT+SPACE. If name of valid function or procedure precedes the parenthesis, then the expected parameters will be prompted to you in a floating panel. As you type the actual parameter, next expected parameter will become bold.



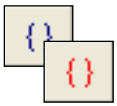
Code Template [CTR+J]

You can insert Code Template by typing the name of the template (for instance, *proc*), then press CTRL+J, and Editor will automatically generate code. Or you can click button from Code toolbar and select template from the list.

You can add your own templates to the list. Just select Tools > Options from drop-down menu, or click Tools Icon from Settings Toolbar, and then select Auto Complete Tab. Here you can enter the appropriate keyword, description, and code of your template.

Auto Correct

Auto Correct corrects common typing mistakes. To access the list of recognized typos, select Tools > Options from drop-down menu, or click Tools Icon from Settings Toolbar, and then select Auto Correct Tab. You can also add your own preferences to the list.



Comment / Uncomment Icon.

Also, Code Editor has feature to comment or uncomment selected block of code by simple click of a mouse, using icons and from Code Toolbar.

Bookmarks

Bookmarks make navigation through large code easier.

CTRL+<number> : Goto bookmark
CTRL+SHIFT+<number> : Set bookmark

Goto Line

Goto Line option makes navigation through large code easier. Select Search > Goto Line from drop-down menu, or use the shortcut CTRL+G.

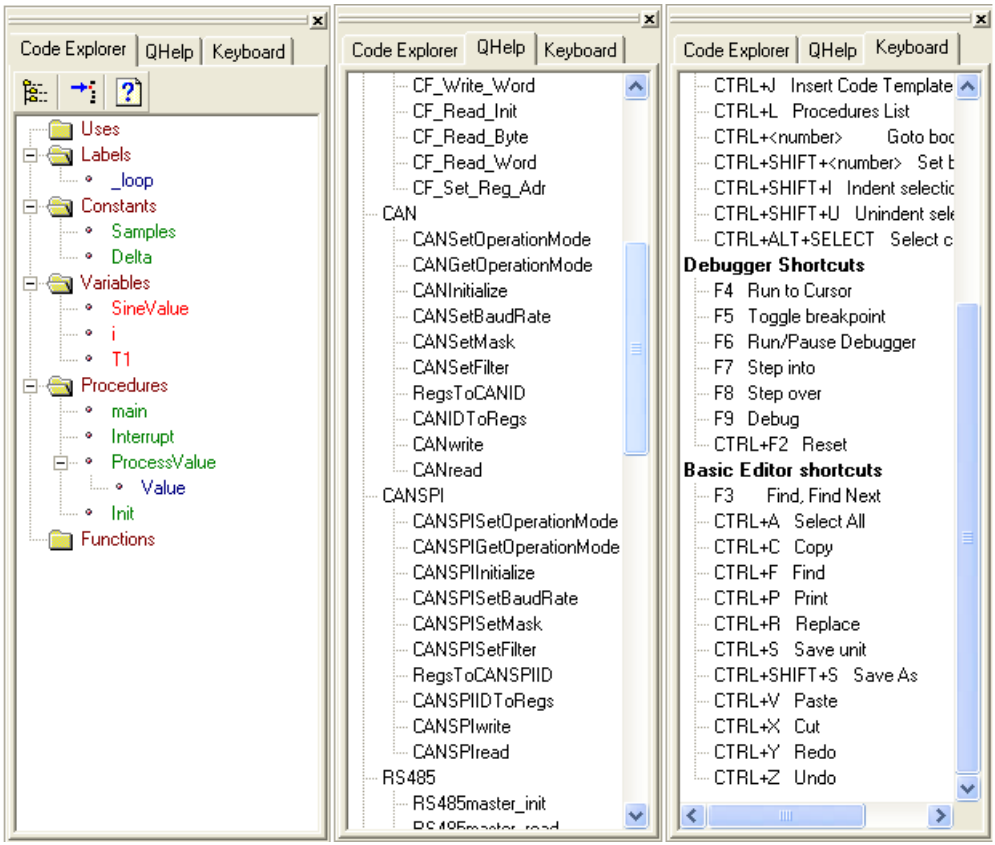
CODE EXPLORER

Find Declaration Icon.

Code Explorer is placed to the left of the main window by default, and gives clear view of every declared item in the source code. You can jump to declaration of any item by right clicking it, or by clicking the Find Declaration icon. To expand or collapse treeview in Code Explorer, use the Collapse/Expand All icon.

Collapse/Expand All Icon.

Also, two more tab windows are available in Code Explorer: Keyboard Tab lists all keyboard shortcuts, and QHelp Tab lists all the available built-in and library functions and procedures, for a quick reference. Double-clicking a routine in QHelp Tab opens an appropriate Help chapter.

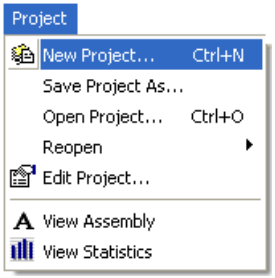


CREATING FIRST PROJECT

New Project Icon.

Step 1

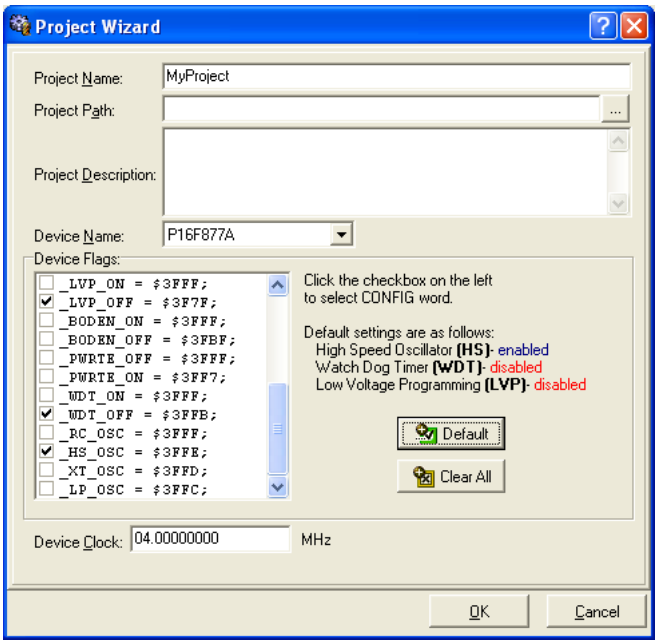
From a drop-down menu, select: Project > New Project, or click New Project icon



Step 2

Fill the New Project Wizard dialog with correct values to set up your new project.

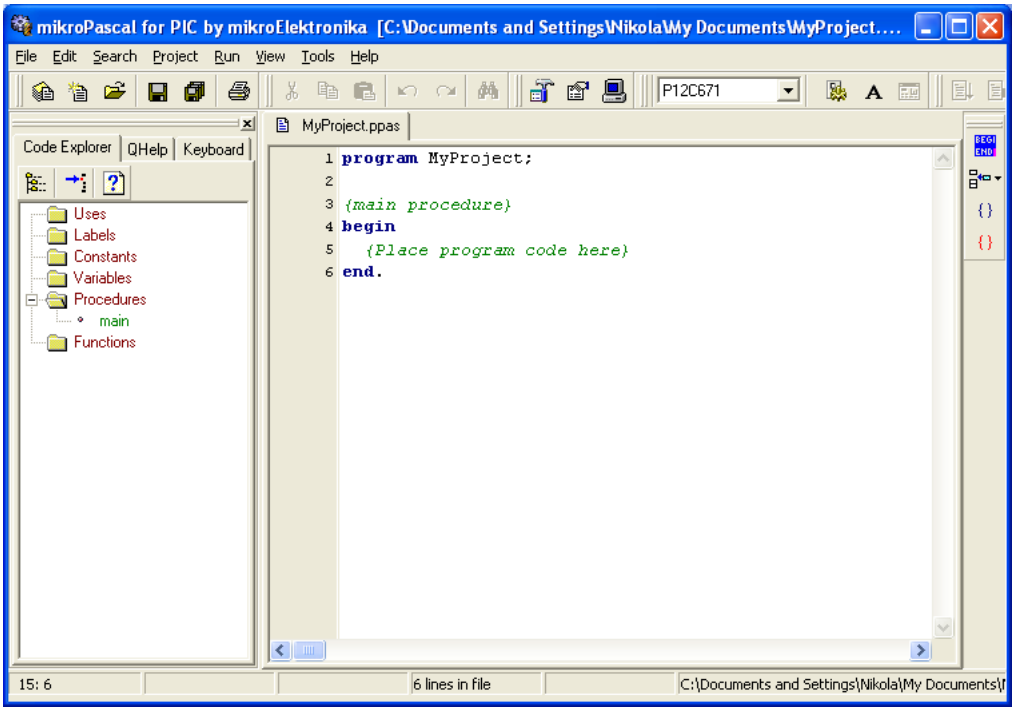
- Select a device for your project from the drop-down menu
- Set configuration bits (Device Flags) by clicking Default push-button.
- Select Device Clock by entering appropriate value in edit box.
- Enter a name for your new project
- Enter project description edit box for closer information about your project
- Enter project path



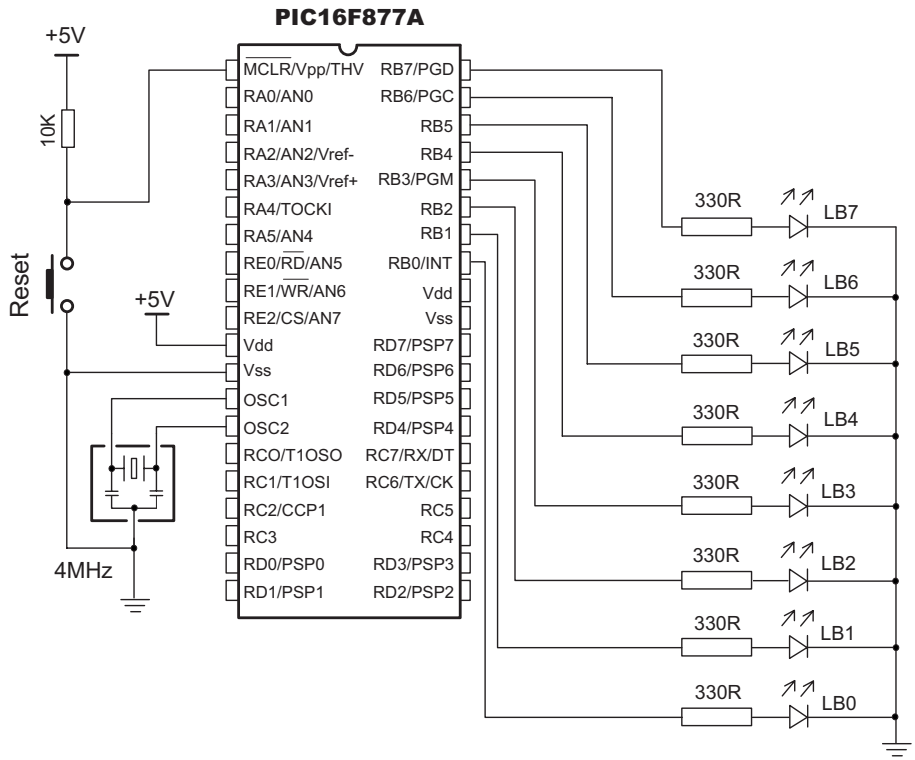
After you have set up your project, select OK push button in New Project Wizard dialog box. mikroPascal will create project for you and automatically open the program file in code editor. Now we can write the source code.

Step 3

After you have successfully created an empty project with New Project Wizard, Code Editor will display an empty program file, named same as your project.



Now we can write the code for this simple example. We want to make LED diode blink once per second. Assuming we have the configuration given in the following figure, LED diodes are connected to PIC16F877 PORTB pins. (it can be any other PIC that has PORTB)



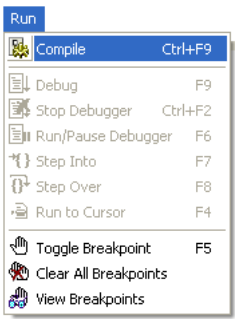
In this configuration, LED will emit light when voltage on pin is high (5V), and will be off when voltage on pin is low (0V). We have to designate PORTD pins as output, and change its value every second. Listing of program is below

```
program My_LED;
begin
  TRISB:=0;                 // configure pins of PORTB as output
  repeat                    // beginning of repeat loop
    begin
      PORTB := $FF;         // turn on diodes on PORTB
      delay_ms(1000);       // wait 1 second
      PORTB := 0;           // turn of diodes on PORTB
      delay_ms(1000);       // wait 1 second
    end;
  until (0=1);              // as this condition is never satisfied
                            // program stays in repeat loop
end.
```

Compile Icon.

Step 4

Before compiling, it is recommended to save the project (menu choice File>Save All). Now you can compile your code by selecting menu Run > Compile, or by clicking the Compile icon.



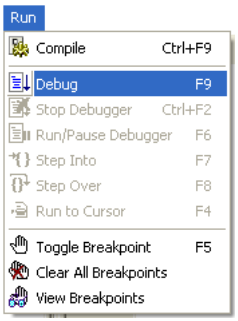
mikroPascal has generated hex file which can be used to program PIC MCU. But before that, let's check our program with the Debugger. Also mikroPascal generates list and assembly files.

Debug Icon.

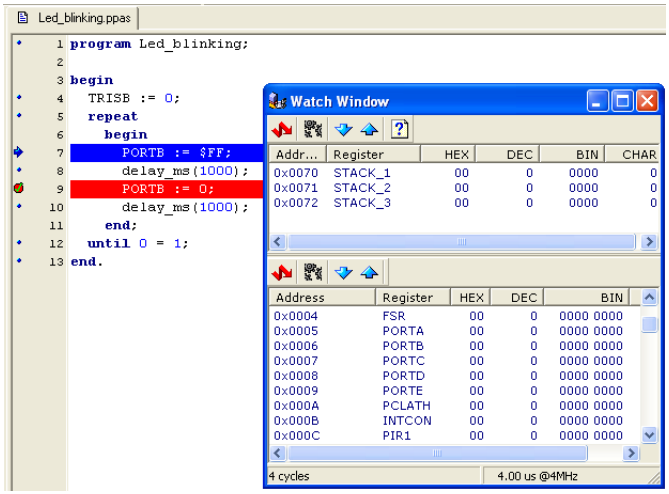
Step 5

After successful compiling, we can use mikroPascal Debugger to check our program behavior before we feed it to the device (PIC16F877 or other). For a simple program such as this, simulation is not really necessary, but it is a requirement for more complex programs.

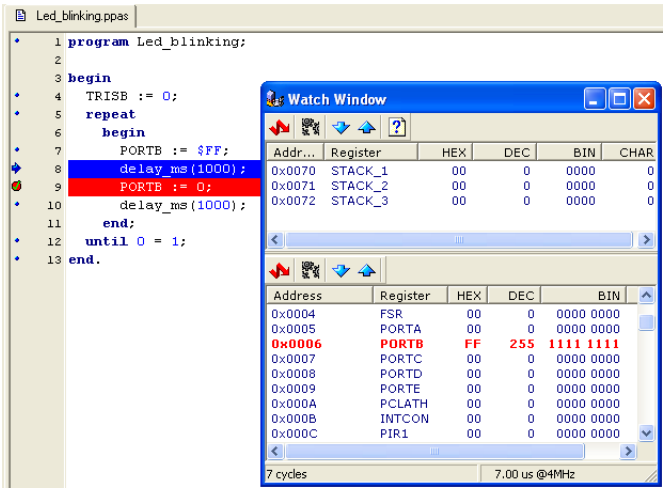
To start the Debugger, select Run > Debug, or click the Debug icon, or simply hit F9.



Upon starting the Debugger, Watch Window appears, and the active line in Code Editor marks the instruction to be executed next. We will set the breakpoint at line 9 by positioning the cursor to that line and toggling the breakpoint (Run > Toggle Breakpoint or F5). See the following image.


Led_blinking.pas
1 program Led_blinking;
2
3 begin
4 TRISB := 0;
5 repeat
6 begin
7 PORTB := $FF;
8 delay_ms(1000);
9 PORTB := 0;
10 delay_ms(1000);
11 end;
12 until 0 = 1;
13 end.
Watch Window
Addr... Register HEX DEC BIN CHAR
0x0070 STACK_1 00 0 0000 0
0x0071 STACK_2 00 0 0000 0
0x0072 STACK_3 00 0 0000 0
Address Register HEX DEC BIN
0x0004 FSR 00 0 0000 0000
0x0005 PORTA 00 0 0000 0000
0x0006 PORTB 00 0 0000 0000
0x0007 PORTC 00 0 0000 0000
0x0008 PORTD 00 0 0000 0000
0x0009 PORTE 00 0 0000 0000
0x000A PCLATH 00 0 0000 0000
0x000B INTCON 00 0 0000 0000
0x000C PIR1 00 0 0000 0000
4 cycles
4.00 us @4MHz

We will use the Step Over option (Run > Step Over or F8) to execute the current program line. Now, you can see the changes in variables, SFR registers, etc, in the Watch Window – items that have changed are marked red, as shown in the image below.


Led_blinking.pas
1 program Led_blinking;
2
3 begin
4 TRISB := 0;
5 repeat
6 begin
7 PORTB := $FF;
8 delay_ms(1000);
9 PORTB := 0;
10 delay_ms(1000);
11 end;
12 until 0 = 1;
13 end.
Watch Window
Addr... Register HEX DEC BIN CHAR
0x0070 STACK_1 00 0 0000 0
0x0071 STACK_2 00 0 0000 0
0x0072 STACK_3 00 0 0000 0
Address Register HEX DEC BIN
0x0004 FSR 00 0 0000 0000
0x0005 PORTA 00 0 0000 0000
0x0006 PORTB FF 255 1111 1111
0x0007 PORTC 00 0 0000 0000
0x0008 PORTD 00 0 0000 0000
0x0009 PORTE 00 0 0000 0000
0x000A PCLATH 00 0 0000 0000
0x000B INTCON 00 0 0000 0000
0x000C PIR1 00 0 0000 0000
7 cycles
7.00 us @4MHz

We could have used Run/Pause (F6) option to execute all the instructions between the active line and the breakpoint (Run > Run/Pause Debugger).

Step 6

Now we can use hex file and feed it to the device (PIC16F877 or other). In order to do so hex file must be loaded in programmer (PIC Flash by mikroElektronika or any other).

For advanced users:

You can set any configuration for PIC MCUs config word(s). Set configuration bits (Device Flags) by clicking appropriate check boxes.

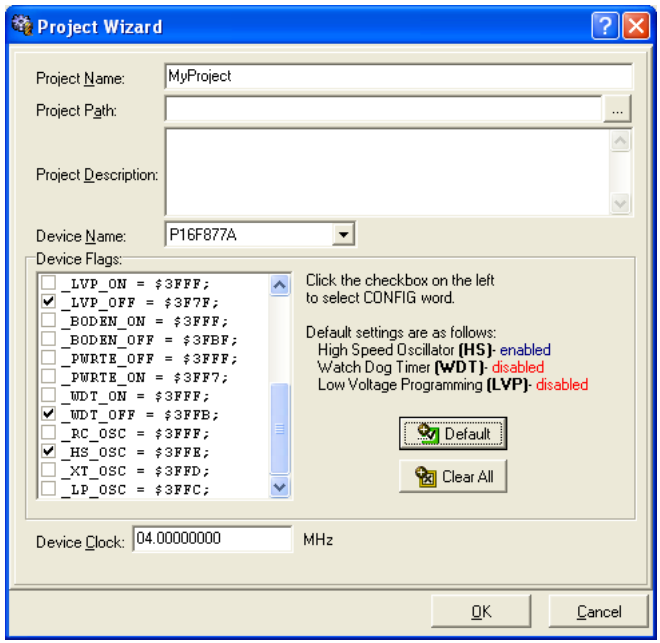
NOTE: If you use internal PLL (Phase Locked Loop), device clock equals 4 x oscillator clock (be sure to enter the appropriate value is in edit box).

PROJECTS

Each application, or project, consists of a single project file and one or more unit files. You can compile source files only if they are part of the project. First and essential step is creating a project.

We will use New Project Wizard to create our new project.

Select Project > New Project from drop-down menu and follow the dialog:



(select PIC MCU device, device clock, setup configuration bits, set project name, location and description)

Later, if you want to change some project settings, select Project > Edit from drop-down menu. To save your project , select Project > Save All from drop-down menu. To save your project under different name, select Project > Save Project As from drop-down menu. To open a project, select Project > Open, or Project > Reopen from drop-down menu.

When you create new project, mikroPascal automatically creates an empty main unit file in which you'll write your source code.

Managing Source Files

Source files created in mikroPascal have the extension ppas. By default, main unit file is named same as the project.

Location of the main unit source file and other project information are stored in project file with extension ppp.

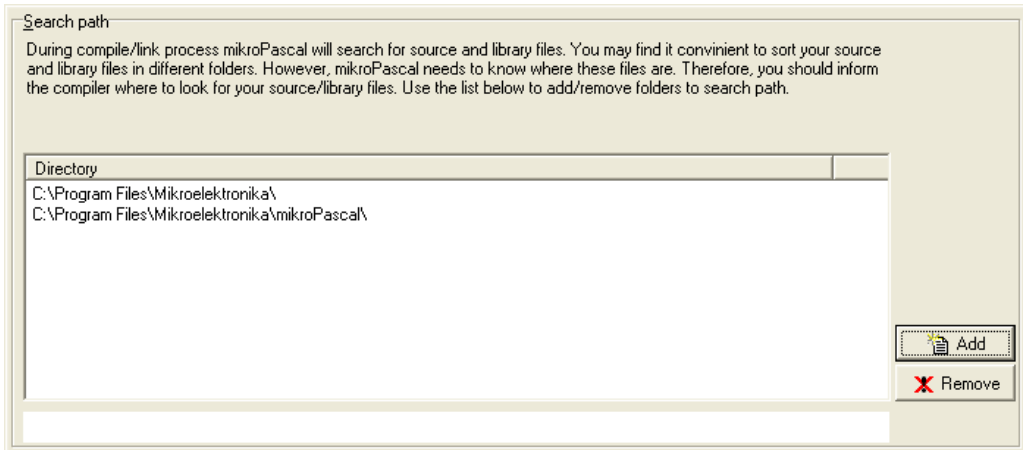
Creating Main Unit File

Main unit file is created simultaneously with the project and is named same as the project, with extension ppas. You should not change the name of this file as mikroPascal might not be able to compile it. Project file and main unit file must be saved in the same folder.

Creating a New Unit File

Select File > New unit from drop-down menu, or press CTRL+N, or click the New File icon. A new tab will open, named "Untitled1". This is your new unit file. Select File > Save As from drop-down menu to name it the way you want.

Keyword uses instructs compiler which unit beside main unit should be compiled. Units other than main must be in same folder with project file or in folder specified by search path. Search path can be configured by selecting menu choice Tools > Options from drop-down menu and then tab window Advanced.


Search path
During compile/link process mikroPascal will search for source and library files. You may find it convinient to sort your source and library files in different folders. However, mikroPascal needs to know where these files are. Therefore, you should inform the compiler where to look for your source/library files. Use the list below to add/remove folders to search path.
Directory
C:\Program Files\Mikroelektronika\
C:\Program Files\Mikroelektronika\mikroPascal\
Add
Remove

Open File Icon.

Opening an Existing File

Select File > Open from drop-down menu, or press CTRL+O, or click the Open File icon. The Select Input File dialog opens. In the dialog, browse to the location of the file you want to open and select it. Click the Open button.
The selected file is displayed in its own tab. If the selected file is already open, its current Editor tab will become active.

Print File Icon.

Printing an Open File

Make sure that window containing the file you want to print is the active window. Select File > Print from drop-down menu, or press CTRL+P, or click the Print icon. In the Print Preview Window, set the desired layout of the document and click the OK button. The file will be printed on the selected printer.

Save File Icon.

Saving File

Make sure that window containing the file you want to save is the active window. Select File > Save from drop-down menu, or press CTRL+S, or click the Save icon. The file will be saved under the name on its window.

Saving File Under a Different Name

Make sure that window containing the file you want to save is the active window. Select File > Save As from drop-down menu, or press SHIFT+CTRL+S. The New File Name dialog will be displayed.
In the dialog, browse to the folder where you want to save the file.
In the File Name field, modify the name of the file you want to save.

Closing a File

Click the Save button. Make sure that tab containing the file you want to close is the active tab. Select File > Close from drop-down menu, or right click the tab of the file you want to close in Code Editor. If the file has been changed since it was last saved, you will be prompted to save your changes.

Compile Source Code

Compile Icon.

When you have created the project and written the source code, you will want to compile it. Select Run > Compile from drop-down menu, or click Compiler Icon from Compiler Toolbar.

Progress bar will appear to inform you about the status of compiling. If no errors are encountered, mikroPascal will produce hex file, assembly file, and list for the appropriate PIC MCU.

DEBUGGER

Source-level Debugger is integral component of mikroPascal development environment. It is designed to simulate operations of Microchip Technology's PIC MCU's and to assist users in debugging Pascal software written for these devices.

Debug Icon.

Debugger simulates program flow and execution of instruction lines, but does not fully emulate PIC device behavior: it does not update timers, interrupt flags, etc. Jump to interrupt is performed by clicking the Interrupt icon .

Debug Icon.

After you have successfully compiled your project, you can run Debugger by selecting Run > Debug from drop-down menu, or by clicking Debug Icon . Starting the Debugger makes more options available: Step Into, Step Over, Run to Cursor etc. Line that is to be executed is color highlighted (blue).

Debug [F9]
Starts Debugger.

Step Into [F7]
Execute the current Pascal instruction (single or multiple cycle instructions) and then halt. After execution, all windows are updated. If the instruction is a procedure or function call, execute it enters routine and halt at the first following instruction after the call.

Step Over [F8]
Execute the current Pascal instruction (single or multiple cycle instructions) then halt. If the instruction is a procedure or function call, execute the called routine and halt at the instruction following the call.

Run to cursor [F4]
Executes all instructions between the current instruction and the cursor position.

Toggle Breakpoints [F5]
Toggle breakpoint at current cursor position.

Debug Icon.

Run/Pause Debugger [F6]

Run or pause Debugger.

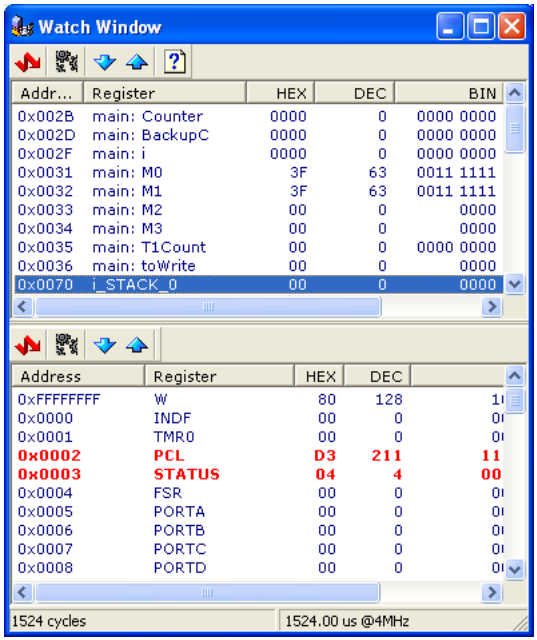
Run > View Breakpoints

Debug Icon.

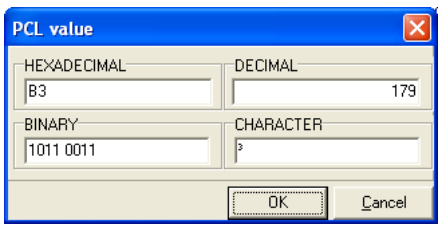
Invoke breakpoints window, with list of breakpoints. Double clicking item in window list locates breakpoint.

Watch Window

Watch Window allows you to monitor program items while running your program. It displays variables and special function registers of PIC MCU, their addresses and values. Values are updated as you go through the simulation. See the image below.



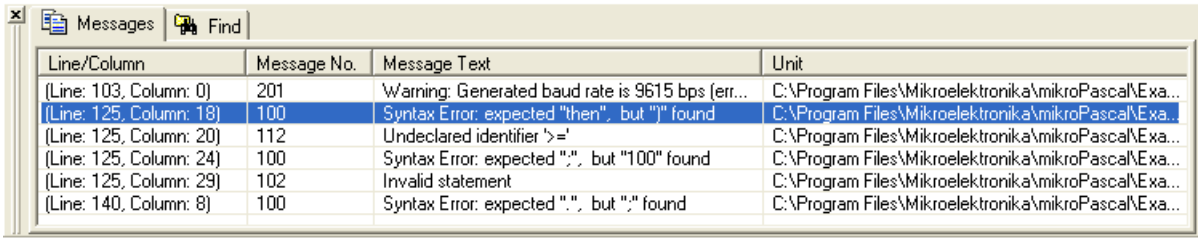
Double clicking one of the items opens a window in which you can assign new value to the selected variable or register.



ERROR WINDOW

In case that errors were encountered during compiling, compiler will report them and won't generate a hex file. Error Window will be prompted at the bottom of the main window.

Error Window is located under message tab, and displays location and type of errors compiler has encountered. Compiler also reports warnings, but these do not affect generating hex code. Only errors can interefere with generation of hex.


Messages
Find
Line/Column
Message No.
Message Text
Unit
(Line: 103, Column: 0) 201 Warning: Generated baud rate is 9615 bps (err... C:\Program Files\Mikroelektronika\mikroPascal\Exa...
(Line: 125, Column: 18) 100 Syntax Error: expected "then", but ")" found C:\Program Files\Mikroelektronika\mikroPascal\Exa...
(Line: 125, Column: 20) 112 Undeclared identifier '>=' C:\Program Files\Mikroelektronika\mikroPascal\Exa...
(Line: 125, Column: 24) 100 Syntax Error: expected ";", but "100" found C:\Program Files\Mikroelektronika\mikroPascal\Exa...
(Line: 125, Column: 29) 102 Invalid statement C:\Program Files\Mikroelektronika\mikroPascal\Exa...
(Line: 140, Column: 8) 100 Syntax Error: expected ".", but ";" found C:\Program Files\Mikroelektronika\mikroPascal\Exa...

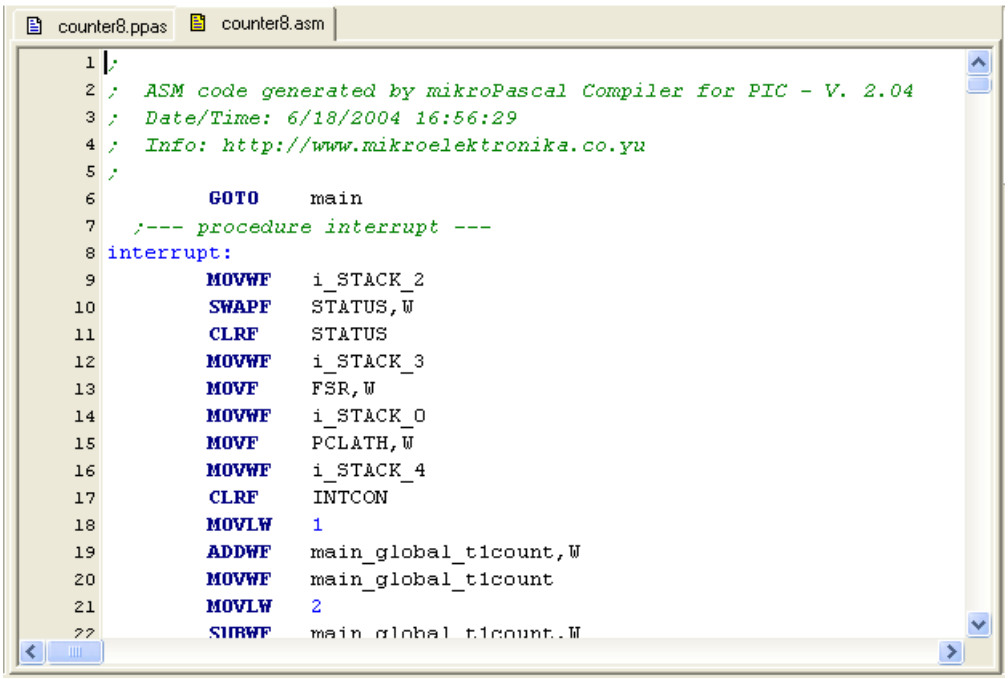
Double clicking the message line in Error Window results in highlighting the line of source code where the error took place.

ASSEMBLY VIEW

Assembly Icon.

After compiling your program in mikroPascal, you can click toolbar Assembly icon or select Project > View Assembly from drop-down menu to review generated assembly code in a new tab window. Assembly is human readable with symbolic names. All physical addresses and other information can be found in Statistics or in list file.

If program is not compiled and there is no assembly file, starting this option will compile your code and then display assembly.



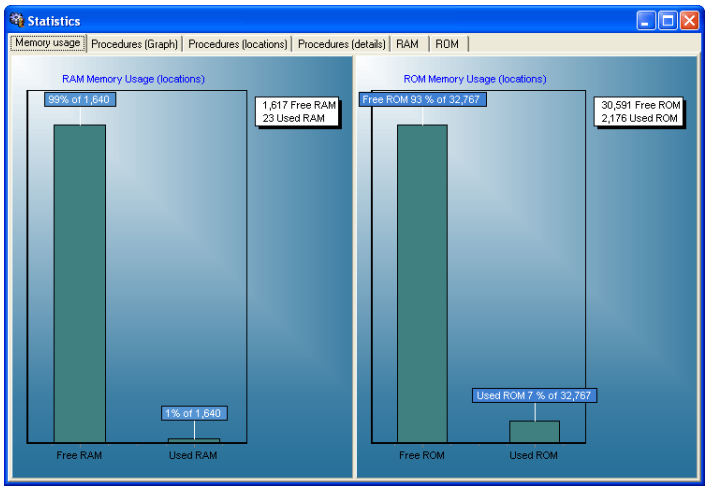
STATISTICS

Statistics Icon.

After successful compiling, you can review statistics on your code. Select Project > View Statistics from drop-down menu, or click the Statistics icon. There are five tab windows:

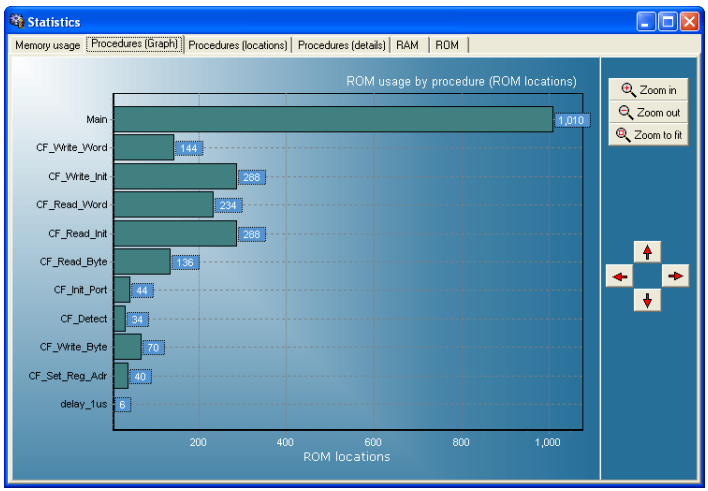
Memory Usage Window

Provides overview of RAM and ROM memory usage in form of histogram.



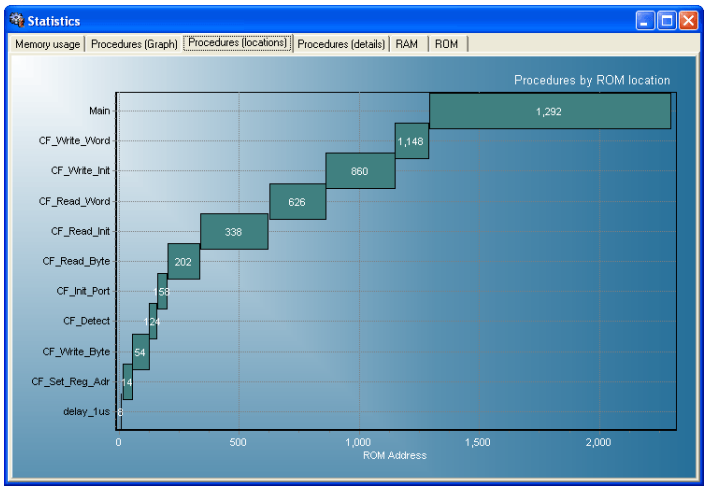
Procedures (Graph) Window

Displays procedures and functions in form of histogram, according to their memory allotment.



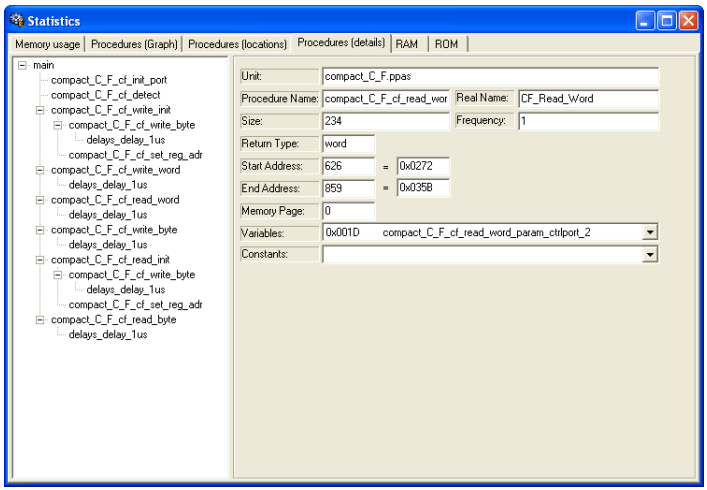
Procedures (Locations) Window

Displays how procedures and functions are located in microcontroller's memory.



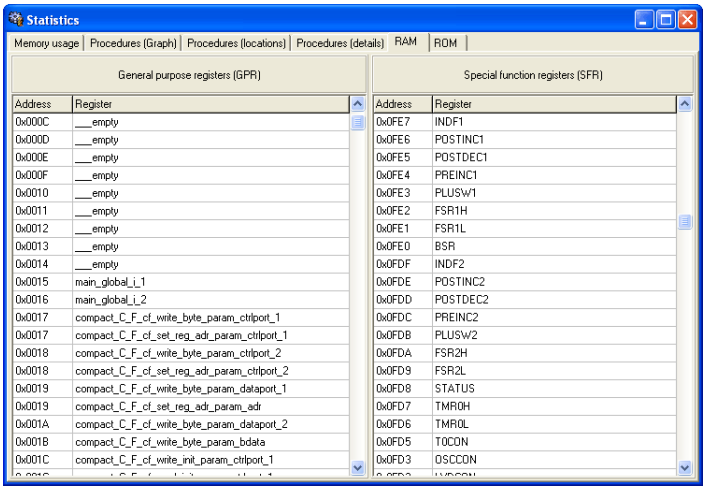
Procedures (Details) Window

Displays complete call tree, along with details for each procedure and function: size, start and end address, frequency in program, return type, etc.



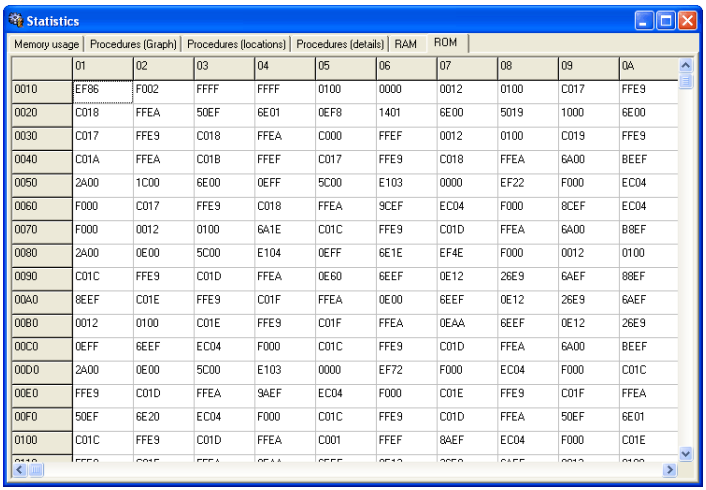
RAM Window

Summarizes all GPR and SFR registers and their addresses. Also displays symbolic names of variables and their addresses.



ROM Window

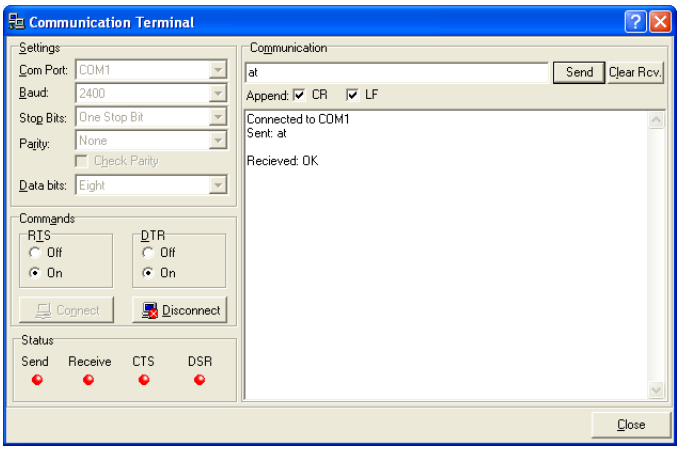
Lists op-codes and their addresses in form of a human readable hex code.



INTEGRATED TOOLS

USART Terminal

mikroPascal includes USART (Universal Synchronous Asynchronous Receiver Transmitter) communication terminal for RS232 communication. You can launch it from drop-down menu Tools > Terminal or by clicking the icon .

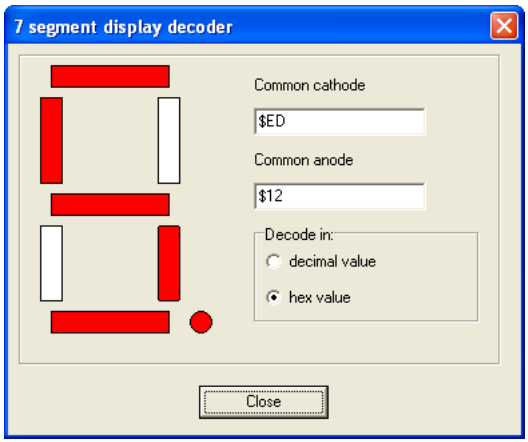


ASCII Chart

ASCII Chart is a handy tool, particularly useful when working with LCD display. You can launch it from drop-down menu Tools > ASCII chart.

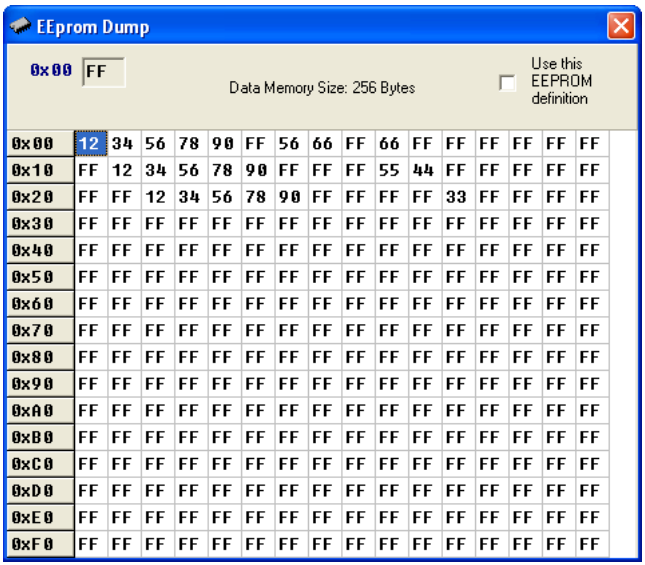
7 Segment Display Decoder

7seg Display Decoder is a convenient visual panel which returns decimal/hex value for any viable combination you would like to display on 7seg. Click on the parts of 7 segment image to the left to get the desired value in the edit boxes. You can launch it from drop-down menu Tools > 7 Segment Display.



EEPROM Editor

EEPROM Editor allows you to easily manage EEPROM of PIC microcontroller.



0x00	12	34	56	78	90	FF	56	66	FF	66	FF	FF	FF	FF	FF	FF
0x10	FF	12	34	56	78	90	FF	FF	FF	55	44	FF	FF	FF	FF	FF
0x20	FF	FF	12	34	56	78	90	FF	FF	FF	FF	33	FF	FF	FF	FF
0x30	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0x40	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0x50	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0x60	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0x70	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0x80	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0x90	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0xA0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0xB0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0xC0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0xD0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0xE0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0xF0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF

KEYBOARD SHORTCUTS

Complete list of keyboard shortcuts is available from Code Explorer window, tab Keyboard.

IDE Shortcuts

F1	Help
CTRL+N	New Unit
CTRL+O	Open
CTRL+F9	Compile
CTRL+F11	Code Explorer on/off
CTRL+SHIFT+F5	View breakpoints

Advanced Editor shortcuts

CTRL+SPACE	Code Assistant
CTRL+SHIFT+SPACE	Parameters Assistant
CTRL+D	Find declaration
CTRL+G	Goto line
CTRL+J	Insert Code Template
CTRL+<number>	Goto bookmark
CTRL+SHIFT+<number>	Set bookmark
CTRL+SHIFT+I	Indent selection
CTRL+SHIFT+U	Unindent selection
CTRL+ALT+SELECT	Select columns

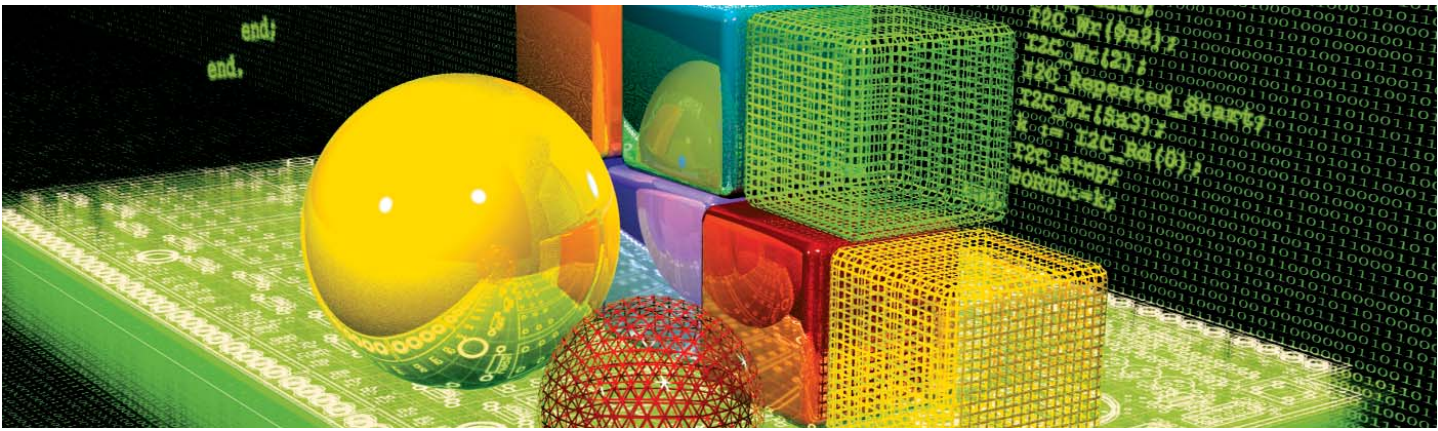
Debugger Shortcuts

F4	Run to Cursor
F5	Toggle breakpoint
F6	Run/Pause Debugger
F7	Step into
F8	Step over
F9	Debug
CTRL+F2	Reset

Basic Editor shortcuts

F3	Find, Find Next
CTRL+A	Select All
CTRL+C	Copy
CTRL+F	Find
CTRL+P	Print
CTRL+R	Replace
CTRL+S	Save unit
CTRL+SHIFT+S	Save As
CTRL+V	Paste
CTRL+X	Cut
CTRL+Y	Redo
CTRL+Z	Undo

CHAPTER 2



mikroPascal Reference

Why Pascal in the first place? Because it is legible, friendly, easy-to-learn structured language, and still has the power and flexibility needed to program microcontrollers. Whether you had any previous programming experience, you will find that writing programs in mikroPascal is very easy. This chapter will help you learn or recollect Pascal syntax, along with the specifics of programming PIC microcontrollers.

IDENTIFIERS

Identifiers are names used for referencing the stored values, such as variables and constants. Every program, procedure, and function must be identified (hence the term) by an identifier.

Rules

Valid identifier:
1. must begin with a letter of English alphabet or possibly the underscore (_)
2. can be followed by alphanumeric characters and the underscore (_)
3. may not contain special characters:
~ ! @ # $ % ^ & * ( ) + ` - = { } [ ] : " ; ' < > ? , . / | \

mikroPascal is not case sensitive. First, FIRST, and fIrST are an equivalent identifier.

Note

Elements ignored by the compiler include spaces, new lines, and tabs. All these elements are collectively known as the white space. White space serves only to make the code more legible; it does not affect the actual compiling.

Several identifiers are reserved in mikroPascal - you cannot use them as your own identifiers. Please refer to Kewords. Also, mikroPascal has several pre-defined identifiers. Pre-defined identifiers are listed in the chapter Library Functions and Procedures.

Examples

```
// Valid identifier examples

temperature_V1
Pressure
no_hit
dat
sum
vtext

// Some invalid identifier examples

7temp         // cannot begin with a numeral
%higher       // cannot contain special characters
xor           // cannot match reserved word
j23.07.04     // cannot contain special characters
```

KEYWORDS

The following keywords (reserved words) cannot be redefined or used as identifiers.

absolute	mod
and	not
array	or
asm	otherwise
begin	print
boolean	procedure
break	program
case	real
char	read
continue	record
const	repeat
div	shl
do	shr
downto	step
else	string
end	then
for	to
function	type
goto	unit
if	until
implementation	uses
in	var
interrupt	while
is	with
label	xor

In mikroPascal, all SFR (Special Function Registers) are defined as global variables and represent special reserved words that cannot be redefined. For example - TMR0, PCL, STATUS, etc.

Also, mikroPascal has a number of predefined identifiers (refer to Library Routines). These can be replaced by your own definitions, but that would impede the functionality of mikroPascal.

DATA TYPES

Type determines the allowed range of values for variable, and which operations may be performed on it. It also determines the amount of memory used for one instance of that variable.

Simple

Type	Size	Range of values
byte	8-bit	0 .. 255
char*	8-bit	0 .. 255
word	16-bit	0 .. 65535
short	8-bit	-128 .. 127
integer	16-bit	-32768 .. 32767
longint	32-bit	-2147483648 ..147483647

* char type can be treated as byte type in every aspect

Structured

Array represents an indexed collection of elements of the same type, often called the base type. Base type can be any simple type.

String represents a sequence of characters. It is an array that holds characters and the first element of string holds the number of characters (max number is 255).

Sign

Sign is important attribute of data types, and affects the way variable is treated by the compiler.

Unsigned can hold only positive numbers:

byte	0 .. 255
word	0 .. 65535

Signed can hold both positive and negative numbers:

short	-128 .. 127
integer	-32768 .. 32767
longint	-2147483648 .. 214748364

Array

Array is a set of data stored in consecutive memory locations. Defining an array and manipulating its elements is simple. Elements of array are always of same data type (any simple).

```
var  days_of_the_week      : array[7]  of byte;
months                     : array[12] of byte;
AD_Conversion_result       : array[10] of word;
```

First declaration above generates 7 variables of byte type. These can be accessed by array name followed by number in the square brackets [] (this number is also known as index). Indexing is zero based, meaning that in our example, index spans numbers from 0 to 6. Instead of byte, you can define array of any other simple type (word, short, integer or longint).

Note that:

```
var something : array[10] of integer;
```

occupies 20 RAM locations (bytes), not 10.

Array and Operators

You can use any kind of operator with array elements - Arithmetic Operators, Logical (Bitwise) Operators, and Relation (Comparison) Operators. Technically, array element is treated as a simple type. Also, instead of a number, index can be any expression with result type of byte. For example:

```
m[a + b] := 90;
m[1] := m[2] + 67;
m[1] := m[2] div m[3];
```

Array and PIC

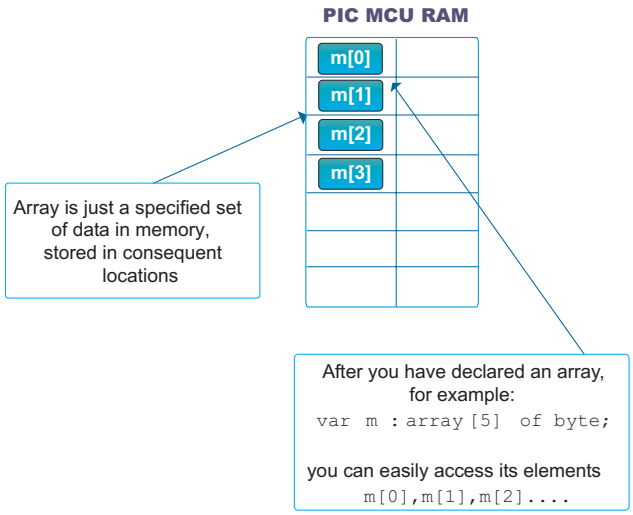
When you declare an array, mikroPascal allocates a certain amount of RAM for it. Elements of array consume consecutive RAM locations; in case of array of bytes, if the address of m[0] is $23, m[1] will be at $24, and so on.

Accessing these elements is almost as fast as accessing any variable of simple type. Instead of byte you can define array of any other simple type (word, short, integer or longint). Don't forget that you are restricted by the amount of free space in PIC RAM memory.

For example:

```
var size : array[10] of longint
```

occupies 40 RAM locations (bytes).


PIC MCU RAM
m[0]
m[1]
m[2]
m[3]
Array is just a specified set of data in memory, stored in consequent locations
After you have declared an array, for example:
var m :array[5] of byte;
you can easily access its elements
m[0],m[1],m[2]....

Example

```
program Array_test;
var     m       :       array[13] of byte;
        j       :       array[5]  of byte;
begin
j[0] := m[3] + 6;
m[4] := m[2] mod 3;
j[2] := not j[0];

if m[0] > 0 then
     m[1] := 9
  else
     begin
       m[1] := 90;
     end;
end.
```

Strings

String represents a sequence of characters. String type is similar to array, but can hold only characters.

```
var M               : string[16];
    Start_message   : string[6];
```

For each string declaration, compiler will reserve the appropriate amount of memory locations. For example, string M will take 16+1 locations; additional memory location is reserved to contain the length of the string. If we assign string literal to variable M, M := 'mik', then:

`M[0]` will be 3 (contains length of the string)
`M[1]` will be 'm'
`M[2]` will be 'i'
`M[3]` will be 'k'

and all the other locations will be undefined.

Strings and assignment

Assignment operator can be used with string variables:

```
var    M : string[20];
       S : string[8];
begin
  M := 'port';      // Assign 'port' to M;
  S := 'port1';     // Assign 'port1' to S;
end.
```

Length

mikroPascal includes a built-in function Length for working with strings:

```
function Length( text : string ) : byte;
```

It returns string length as byte, and is quite useful for handling characters within string:

```
M := 'mikroElektronika';
for i := 1 to Length(M) do
begin
   LCD_Chr(1,i,M[i]);
end;
```

NUMERALS AND CHARACTER STRINGS

Numerals

Numeric constants can be represented in decimal, binary, or hexadecimal number system.

In decimal notation, they are represented as a sequence of digits, without commas or spaces, and can be prefixed with + or - operator to indicate the sign. Values default to positive (67258 is equivalent to +67258).

The dollar-sign prefix indicates a hexadecimal numeral (for example $8F).

The percent-sign indicates a binary numeral (for example %0101).

Example:

```
123          // Decimal
$1fc         // Hex
%101         // Binary
```

Character Strings

Character string, also called a string literal or a string constant, consists of a quoted string. Separators can be used only within quoted strings. A quoted string is a sequence of up to 255 characters from the extended ASCII character set, written in one line and enclosed by apostrophes.

Quoted string with nothing between the apostrophes is a null string. Apostrophe itself cannot be used as part of the string. For example:

```
'mikroPascal'                { mikroPascal }
''                           { null string }
' '                          { a space }
```

Length of character string is the number of characters it consists of. Character string of length 1 is compatible with the char type. You can assign string literal to a string variable or to array of char.

CONSTANTS

Constant is data whose value cannot be changed during the runtime. Every constant is declared under unique name which must be a valid identifier. It is a good practice to write constant names in uppercase.

In mikroPascal, constants have to be of simple data type (no arrays or strings are allowed).

Example of constant declaration:

```
const MAXVALUE : integer = 237;
```

Constants can be used in any legal expression, but they cannot be assigned a new value. Therefore, they cannot appear on the left side of the assignment operator.

Note

If you frequently use the same value throughout the program and its value is fixed, you should declare it a constant (for example, maximum number allowed is 1000). This is a good practice since the value can be changed simply by modifying the declaration, instead of going trough the entire program and adjusting each instance manually. As simple as this:

```
const MAX : integer = 1000;
```

Constants and PIC

It is important to understand why constants should be used and how this affects the MCU. Using a constant in a program consumes no RAM memory. This is very important due to the limited RAM space (PIC16F877 has 368 locations/bytes).

Examples

```
const  MaxAllowed : byte = 234;
// Single declaration


const K_a : integer = -32766;
      Max : integer = 1000;
      Min : integer = 0;
// Multiple declaration


ADC_Res := ADC_Read(2);
if ADC_Res > Max then
  portb := 1
else
  portb := Min;
// Constants can be used in any expression
```

Examples of invalid use

```
const 7time : byte = 123;
// Wrong constant name, it must be
// a valid identifier


const Max : integer = 1123456;
// Assigned value exceeds the allowed
// range for integer


Max := A;
Max := 123;
// You cannot assign new value to a constant,
// compiler will report an error
```

ARRAY CONSTANTS

Array constants is a set of data stored in ROM memory. Elements of array are always of same data type (any simple). Constant string is same as `array[] of char`.

```
const CTEXT    : array[6]  of char = 'trial1';
const CMONTHS  : array[12] of byte = (1,2,3,4,5,6,7,8,9,10,11,12);
const MATCHVALUE : array[7] of word = (3,4566,56000,324,0,778,876);
```

To declare an array constant holding numerals, enclose the values of the array's elements, separated by commas, in parentheses. For arrays of char type, use a string literal (text enclosed by apostrophes), as shown in the example above. Number of values in parentheses, i.e. a number of characters in apostrophes, must match the specified value in the square brackets.

Note

Accessing elements of array constant is simple, but be careful not to exceed the array range (if index is greater than array size, program won't work correctly). Indexing is zero based, so if you declare:

```
const width : array[3] of byte = (23,5,67);
```

then `width[0]` is the first element and index needs to be less or equal than 2.

Special case is array constant of char which holds the size of the array in the first location. Array constant of char is limited to 255 characters.

Constants and Operators

You can use any kind of operator with array constant elements. Technically, array constant element is treated as a simple constant. Also, instead of a number, index can be any expression. But, you cannot assign new value to array constant element. For example:

```
m := CMONTHS[a + b] ;
m := CMONTHS[2] + 67;
vv := vv div CMONTHS[3];
```

Also, it is possible for array constants of type char to copy whole array to variable of same type by simple assignment. For example:

```
const CTXT : array[4] of char = "dota";
var    txt : array[4] of char;
...
txt := CTXT;
```

Array Constants and PIC

Elements of array constant are located in R0M. PIC16 family restricts array constants to 255 elements of byte type, while PIC18 family is limited only by ROM size.

Instead of byte, you can define array of any other simple type (word, short, integer or longint). Don't forget that you are restricted by the amount of PIC ROM memory.

For example:

```
const foo : array[4] of longint = (36, 32442, 1901, 82, 27332724);
```

Example

```
program Array_test;

const      m  : array[3] of byte = (0,1,2);
var        j  : array[5] of byte;
begin
  j[0] := m[3] + 6;
  j[4] := m[2] mod 3;
  j[2] := not j[0];

if m[0] > 0 then
    j[1] := 9
  else
    begin
      j[1] := 90;
    end;
end.
```

VARIABLES

Variable is data whose value can be changed during the runtime. Every variable is declared under unique name which must be a valid identifier. This name is used for accessing the memory location occupied by the variable.

Variable can be seen as a container for data and because it is typed, it instructs the compiler how to interpret the data it holds. For more details refer to Data Types and Type Conversion.

For more information on variables' scope refer to the chapter Scope (Variable Visibility).

In mikroPascal, variable needs to be declared before it can be used. Specifying a data type for each variable is mandatory. Basic syntax for variable declaration is:

```
var variableList: type;
```

where *variableList* is a comma-delimited list of valid identifiers and type can be any valid data type.

For example:

```
var A  : byte; { declare variable A of byte type }
    BB : word; { declare variable BB of word type }
```

or

```
var X, Y     : short;
    I, J, K  : integer;
```

Variables and PIC

Every declared variable consumes part of MCU RAM memory. Data type of variable determines not only the allowed range of values, but also the space variable occupies in RAM memory. Bear in mind that operations using different types of variables take different time to be completed. For example:

Variable A (byte) occupies 1 byte (8 bit) of RAM memory, while variable BB (word) occupies 2 bytes (16 bit) of RAM memory.

Therefore, A := A + A; is faster to execute than BB := BB + BB;

PIC16F877 has 368 RAM memory locations (bytes). Maximum available number of variables can be calculated in the following manner:

B + S + 2 * W + 2 * I + 4 * LI <= 368 RAM memory locations (example for PIC16F877), where :

B - number of variables of type byte
S - number of variables of type short
W - number of variables of type word
I - number of variables of type integer
LI - number of variables of type longint

NOTE: mikroPascal recycles local variable memory space - local variables declared in different functions and procedures share same memory space, if possible.

Additional info

Variable declaration has to be properly placed to have a correct meaning. Variables can be declared in the program block or implementation section of a unit. Variable declaration goes after the uses statement, and must be placed ahead of the keyword begin. You can also declare variables in function or procedure block.

There is no need to declare PIC SFR (Special Function Registers), as they are already declared as global variables of byte type - for example: TMR0, PCL, STATUS, PORTA, TRISA, etc. These variables may be used anywhere within the code.

For closer information on how to use variables and build valid expressions refer to the chapter Operators.

Examples

```
program TRIAL
uses other;

// You can declare variables in the program block
var tA, tD, tF : integer;
    tR, tT, tY : word;

begin
  tA := tD and tF;

// STATUS and TMR0 are PIC registers
  tR := STATUS and $03;
  TMR0 := 45;
end.


unit other;

implementation
// You can declare variables in implementation
// section of a unit.
var  Sss, Ddd : longint;
...
end.


function Sum(R : byte) : byte;
  // You can also declare variables in
  // function or procedure block.
var B : char;
    K : byte;

begin
  ...
end;
```

Any valid variable can be used after it has been declared :

```
...

tA := 36;
  // assign new value to the existing variable;
tC := tA + tB;
  // perform any kind of arithmetical or
  // logical operation;

tE := pr_function(1,tA);
  // pass variable to function or procedure,
  // by value or address;

pr_procedure(1,2,tD,tE);

{ use them in conditional and/or
  loop statements and more ... }

case tb of
1: tA := tD + 4;
2: tB := tC + 6;
3:
  begin
    tC := $ff;
  end
else pr_procedure(1,2,tD,tE);
end;

for tA := 0 to 7 do
begin
  tC := tB shr(1);
end;
```

COMMENTS

Comments are text that is added to the code for purpose of description or clarification, and are completely ignored by the compiler.

```
{ All text between a left brace and a right brace
  constitutes a comment. May span multiple lines. }

// Any text between a double-slash and the end of the
// line constitutes a comment. May span one line only.
```

It is a good practice to comment your code, so that you or anybody else can later re-use it. On the other hand, it is often useful to comment out a troublesome part of the code, so it could be repaired or modified later.

mikroPascal Code Editor features syntax color highlighting - it is easy to distinguish comments from the code due to different color, and comments are also italicized.

Example

```
var teC,teB,teA : byte;  // declare variable teC of byte type

begin
teC := 12;               // assign value 12 to variable C

if teA > 0 then
    teC := 9
  else
    teA := teB;
        { you can also comment out part of the
          code you don't want to compile: }

        // E := pr_function(1,2);

        // This function call won't be compiled;
end.
```

EXPRESSIONS

Expression is a construction that returns a value. The simplest expressions are variables and constants, while more complex expressions are constructed from simpler ones using operators, function calls, indexes, and typecasts.

Rules for creating legal expressions are presented in chapter Implicit Conversion and Legal Expressions.

These are all expressions:

```
X             { variable }
15            { integer constant }
Calc(X, Y)    { function call }
X * Y         { product of X and Y }
```

Legal Expressions

We will present in short notice rules for building expressions here. But, we should recollect some information beforehand:

Simple data types include: byte, word, short, integer and longint.

Byte and word types hold only positive values so we'll call them unsigned. Ranges are:

byte 0 .. 255
word 0 .. 65535

Short, integer, and longint types can hold both positive and negative numbers so we'll call them signed. Ranges are:

short -128 .. 127
integer -32768 .. 32767
longint -2147483648 .. 214748364

You cannot mix signed and unsigned data types in expressions with arithmetic or logical operators. You can use explicit conversion though.

```
var Sa : short;
    A : byte;
    Bb : word;
    Sbb : integer;
    Scccc : longint;
...
A := A + Sa;     // compiler will report an error;
A := A and Sa;   // compiler will report an error;

// But you can freely mix byte with word..
          Bb := Bb + (A * A);

// ..and short with integer and longint;
          Scccc := Sbb * Sa + Scccc;
```

You can assign signed to unsigned or vice versa only using the explicit conversion.

```
Sa := short(A);
// this can be done; convert A to short,
// then assign to Sa

Sa := A
// this can't be done,
// compiler will report an error
```

Relation operators can freely be used even when mixing signed and unsigned data. For example:

```
if Sa > B then Sa := 0;
```

Note

Comparing variable or constant to variable or constant will always produce correct results.

Comparing expressions requires a little more attention. For more information refer to the chapter Implicit Conversion and Relation Operators.

DECLARATION AND STATEMENTS

Aside from the `uses` clause and reserved words like `implementation` that demarcate parts of a unit, program consists entirely of declarations and statements, which are organized into blocks.

Declarations

Names of variables, constants, types, procedures, functions, programs and units are called identifiers (numeric constant like 321 is not an identifier).

Identifiers need to be declared before you can use them. Only exceptions are few predefined types, library functions and procedures, PIC MCU SFR (PIC Special Function Registers), and constants; these are understood by the compiler automatically.

Declaration defines an identifier and, where appropriate, allocates memory for it. For example:

```
var Right: word;
```

declares a variable called Right that holds a word value, while:

```
function Each(X, Y : integer) : integer;
```

declares a function called Each which collects two integers as arguments and returns an integer.

Each declaration ends with a semicolon (separator). When declaring several variables, constants, or types at the same time, you need to write the appropriate reserved word only once :

```
var
  Height: integer;
  Description: string[10];
```

The syntax and placement of a declaration depends on the kind of identifier you are defining. In general, declarations take place only at the beginning of a block, or at the beginning of the implementation section of a unit (after the uses clause). Specific conventions for declaring variables, constants, types, functions, and so forth can be found in the appropriate chapters.

Statements

Statements define algorithmic actions within a program. Simple statements - like assignments and procedure calls - can be combined to form loops, conditional statements, and other structured statements. Multiple statements within a block are separated by semicolons (separators). Refer to Implicit Conversion and Assignment.

Simple Statements

Simple statement does not contain any other statements. Simple statements include assignments, and calls to procedures and functions.

Structured Statements

Structured statements are constructed from other statements. Use a structured statement when you want to execute other statements sequentially, conditionally, or repeatedly.

Conditional statements if and case execute at most one of their constituents, depending on a specified criteria.

Loop statements repeat, while, and for execute a sequence of constituent statements repeatedly.

Compound Statements

Compound statement is a sequence of simple or structured statements to be executed in the order in which they are written. The compound statement is bracketed together using the reserved words **begin** and **end**, and its constituent statements are separated by semicolons. For example:

```
begin
  Z := X;
  X := Y;
  Y := Z;
end;
```

Compound statements are essential in contexts where Pascal syntax requires a single statement. In addition to program, function, and procedure blocks, they occur within other structured statements, such as conditionals or loops. For example:

```
begin
  I := SomeConstant;
  while I > 0 do
  begin
        I := I - 1;
  end;
end;
```

You can write a compound statement containing only a single constituent statement. Like parentheses in a complex term, begin and end sometimes serve to disambiguate and make the code more legible.

DIRECTIVES

Directives are words of special significance for the mikroPascal, but unlike other reserved words, appear only in contexts where user-defined identifiers cannot occur.

You cannot define an identifier that looks exactly like a directive.

Overview

Directive	Meaning
Absolute	specifies the exact location of variable in RAM
Org	specifies the exact location of routine in ROM

Absolute directive specifies the starting address in RAM for variable (if variable is multi-byte, higher bytes are stored at consecutive locations).

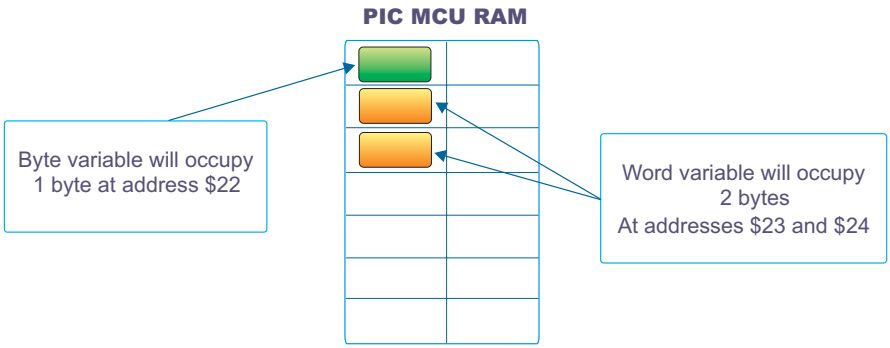
Org directive specifies the starting address of routine in ROM. For PIC16 family, routine must fit in one page - otherwise, compiler will report an error.

Directive `absolute` is appended to the declaration of variable:

```
var rem : byte; absolute $22;
// Variable will occupy 1 byte at address $22;

    dot :  word; absolute $23;
// Variable will occupy 2 bytes at addresses $23 and $24;
```

Directive `org` is appended to the declaration of routine:

```
procedure initial(var attrib : byte); org $200;
begin
// Procedure will start at address $200;
...
end;
```



Important

We recommend careful use of absolute directive, because you may overlap two variables by mistake. For example:

```
var  Ndot : byte; absolute $33;
// Variable will occupy 1 byte at address $33;

     Nrem : longint; absolute $30;
// Variable will occupy 4 bytes at $30, $31, $32, $33,
// so changing Ndot changes Nrem highest
// byte at the same time;
```

Runtime Behavior

mikroPascal uses internal algorithm to distribute variables within RAM. If there is a need to have variable at specific predefined address, use the directive absolute. Also if, for some reason, you want to overlap existing variables, use the directive absolute.

Example

```
program lite;
// example for P16F877A
var image_trisa: byte; absolute 133;

function getnext(data : byte) : byte; org $400;
begin
  result := data + 1;
end;

begin
  image_trisa:=$ff;
  porta := getnext(portb);
end.
```

PROCEDURES AND FUNCTIONS

Procedures and functions, collectively referred to as routines, are self-contained statement blocks that can be called from different locations in a program. Function is a routine that returns a value when it is executed. Procedure is a routine that does not return a value.

Once these routines have been defined, you can call them once or multiple times. Procedure is called upon to perform a certain task, while function is called to compute a certain value. Function calls, because they return a value, can be used as expressions in assignments and operations.

For controlling the program flow, you can use Exit statement to break out from routine.

Procedures

Procedure declaration has the form:

```
procedure procedureName(parameterList);
  localDeclarations
begin
  statements
end;
```

where *procedureName* is any valid identifier, *statements* is a sequence of statements that are executed upon the calling the procedure, and (*parameterList*) and *localDeclarations* are optional declaration of variables and/or constants.

```
procedure pr1_procedure(par1,par2: byte; var vp1,vp2: byte);
var locS : byte;

begin
  par1 := locS + par1 + par2;
  vp1 := par1 or par2;
  vp2 := locS xor par1;
end;
```

par1 and *par2* are passed to the procedure by the value, but variables marked by keyword **var** are passed by the address.

This means that the procedure call

```
pr1_procedure(tA, tB, tC, tD);
```

passes tA and tB by the value: it first creates par1 := tA; and par2 := tB; then manipulates par1 and par2 so that tA and tB remain unchanged;

passes tC and tD by the address: whatever changes are made upon vp1 and vp2 are also made upon tC and tD.

Functions

Function declaration is similar to procedure declaration, except it has a specified return type and a return value. Function declaration has the form:

```
function functionName(parameterList): returnType;
  localDeclarations
begin
  statements
end;
```

where *functionName* is any valid identifier, *returnType* is any simple type, *statements* is a sequence of statements to be executed upon calling the function, and *(parameterList)* and *localDeclarations* are optional declarations of variables and/or constants.

In mikroPascal, function name cannot be used for returning a value of a function (like in standard Pascal). Instead, use the keyword Result to assign the return value of a function. Treat it not as a shortcoming, but as a way of making the code more legible.

Example:

```
function some_function(par1: byte; par2: word) : word;
var locS : word;

begin
  locS := par1 + par2;
  Result := locS;
end;
```

Function Calls

As functions return a value, function calls are technically expressions. For example, if you have defined a function called Calc, which collects two integer arguments and returns an integer, then the function call `Calc(24, 47)` is an integer expression. If I and J are integer variables, then `I + Calc(J, 8)` is also an integer expression. Here are a few examples of function calls:

```
Sum(tA,63)
Maximum(147,J)
GetValue
```

Important

Note that cross-calling and recursive calls are forbidden in mikroPascal. Cross-calling is an instance of procedure A calling procedure B, and then procedure B calling procedure A. Recursive call is an instance of procedure or function calling itself. Compiler will report error if cross-calling or recursive calls are encountered in the code.

mikroPascal has several built in function and predefined Library Functions and Procedures. For example, procedure `interrupt` is the interrupt service routine.

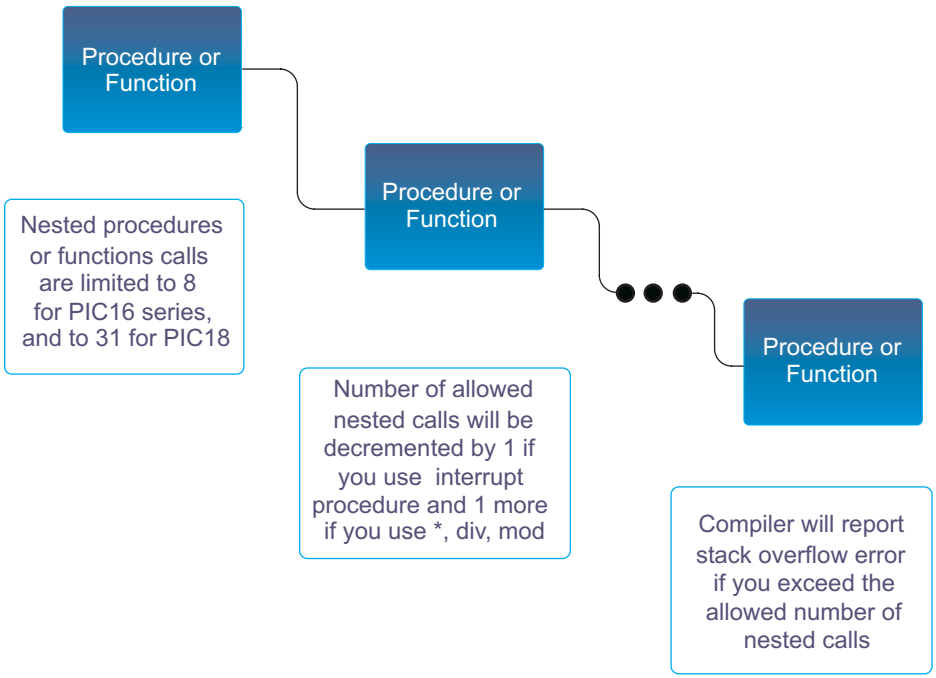
Nested calls are limited to 8-level depth for PIC16 series and 31-level depth for PIC18 series. Nested call represent call of another function or procedure within a function or procedure. For closer information refer to the chapter PIC Specifics.

Example

```
function mask(var num : byte): byte;

begin                          // this function returns code
  case num of                  // for specific digit
    0 : result := $3F;         // for common cathode 7 seg. display
    1 : result := $06;
    2 : result := $5B;
    3 : result := $4F;
    4 : result := $66;
    5 : result := $6D;
    6 : result := $7D;
    7 : result := $07;
    8 : result := $7F;
    9 : result := $6f;
  end;                         // case end
end;
```

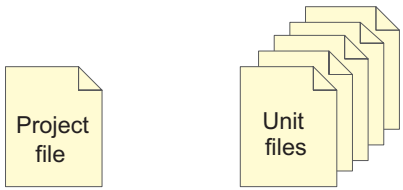
Example of Stack Overflow

```
program Stack_overflow;
procedure interrupt;
begin
  nop;
end;

procedure proc0;
begin
  nop;
end;

procedure proc1;
begin
  proc0;
end;

procedure proc2;
begin
  proc1;
end;

procedure proc3;
begin
  proc2;
end;

procedure proc4;
begin
  proc3;
end;

procedure proc5;
begin
  proc4;
end;

procedure proc6;
begin
  proc5;
end;

procedure proc7;
begin
  proc6;
end;

begin
  proc7;
end.
```


Procedure or Function
Procedure or Function
Procedure or Function
Nested procedures or functions calls are limited to 8 for PIC16 series, and to 31 for PIC18
Number of allowed nested calls will be decremented by 1 if you use interrupt procedure and 1 more if you use *, div, mod
Compiler will report stack overflow error if you exceed the allowed number of nested calls

UNITS

Each project consists of a single project file, and one or more unit files. To build a project, the compiler needs either a source file or a compiled unit file for each unit.



Every project consist of single project file and one or more units files

Each unit is stored in its own file and compiled separately; compiled units are linked to create an application.

Units allow you to:

- Break large programs into modules that can be edited separately.
- Create libraries that can be used in different programs.
- Distribute libraries to other developers without disclosing the source code.

In mikroPascal programming, all source code including the main program is stored in .ppas files.

If you perform circular unit references, compiler will give a warning. A simple instance of circular unit references would be, for example, situation in which Unit1 uses Unit2, but in the same time it is specified that Unit2 uses Unit1.

Newly created blank unit contains the following :

```
unit Unit1;

implementation

end.
```

Unit Influence on Scope (Visibility)

mikroPascal variables defined at the beginning of the unit are global hidden variables. When you declare an identifier in the implementation portion of a unit, you cannot use it outside the unit, but you can use it in any block or procedure defined within the unit. Refer to chapter Scope (Variable Visibility) for more details.

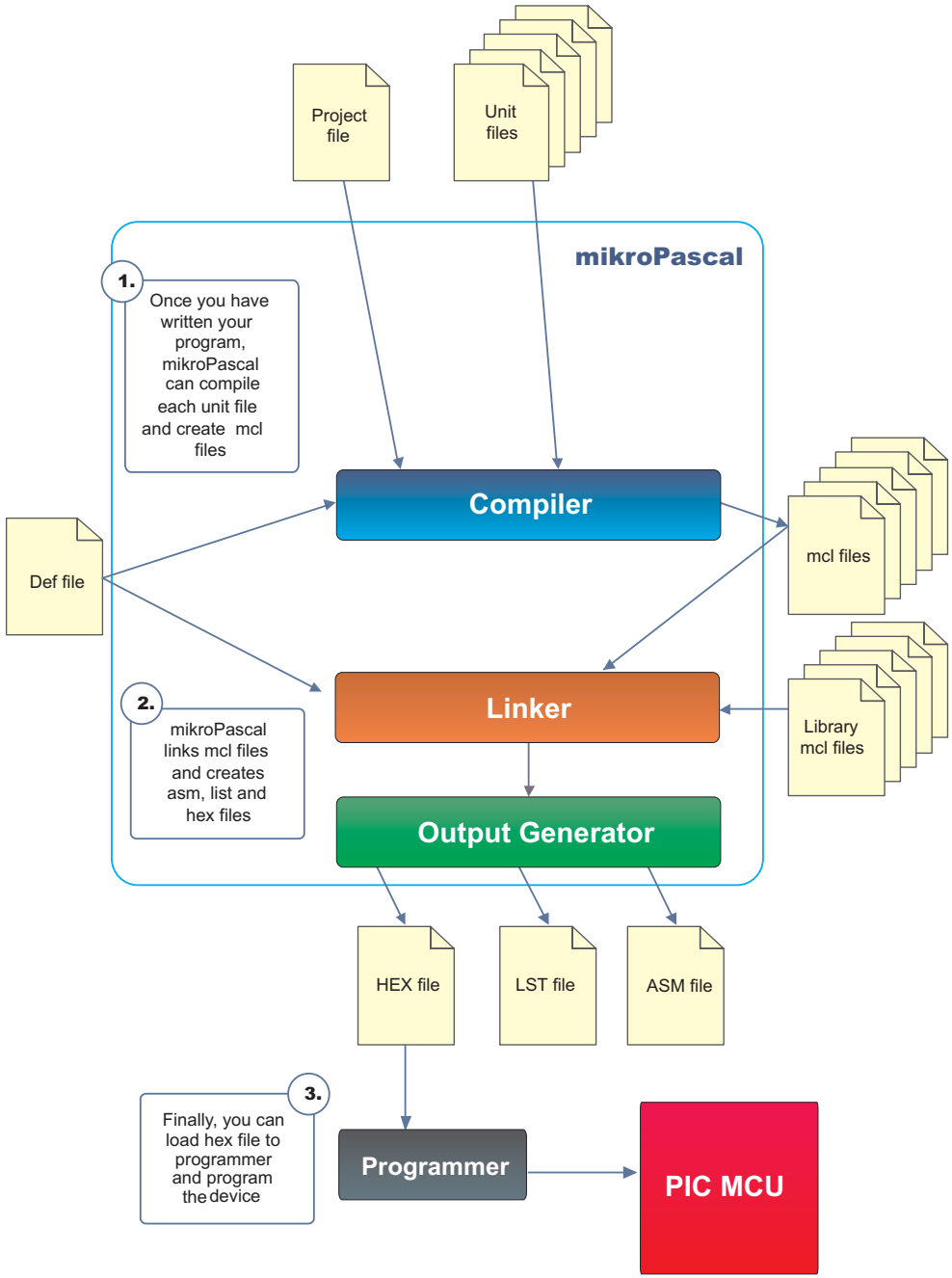
Main Unit File

mikroPascal application has one main unit file and none or more unit files. All source files have the same extension (ppas). Main unit file is identified by the keyword program at the beginning; other unit files include the keyword unit at the beginning and the keyword implementation.

Example

```
program Project1;
uses additional;
var tA, tB : word;

begin
  tA := sqrt(tb);
end.
```

Keyword uses instructs the compiler which file to compile. If you want to include a unit, add the keyword uses followed by the name of the file. The example above includes the unit `additional.ppas` in the program file.


Project file
Unit files
mikroPascal
1.
Once you have written your program, mikroPascal can compile each unit file and create mcl files
Compiler
Def file
mcl files
2.
mikroPascal links mcl files and creates asm, list and hex files
Linker
Library mcl files
Output Generator
HEX file
LST file
ASM file
3.
Finally, you can load hex file to programmer and program the device
Programmer
PIC MCU

SCOPE (IDENTIFIER VISIBILITY)

Scope, or identifier visibility, determines if identifier can be referenced in certain part of the program code. Location of identifier declaration in the code determines its scope. Identifiers with narrower scope - especially identifiers declared in functions and procedures - are sometimes called local, while identifiers with wider scope are called global.

All function and procedures are visible in the whole project, and they are visible in any part of the program or any unit. Constants not local for a procedure or function are also visible in the whole project. Local constants are visible only in procedure or function body in which they are declared.

Rules for determining the variable identifier scope are summarized below:

- If the identifier is declared in the declaration of a main unit, it is visible from the point where it is declared to the end of the unit.

- If the identifier is declared in the declaration of function, or procedure, its scope extends from the point where it is declared to the end of the current block, including all blocks enclosed within that scope.

- If the identifier is declared in the implementation section of a unit, but not within the block of any function or procedure, its scope extends from the point where it is declared to the end of the unit. The identifier is available to any function or procedure in the unit.

PIC SFR (Special Function Registers) such as TMR0, PORTA, etc, are implicitly declared as global variables of byte type. Their scope is the entire project and they are visible in any part of the program or any unit.

Blocks and Scope

Blocks allow a single identifier, such as a variable name, to have different meanings in different parts of the program. Each block is part of the declaration of a program, function, or procedure; each program, function, or procedure declaration has one block.

A block consists of:

```
declarations

begin
  statements
end;
```

Declarations and statements are organized into blocks, which define local name-spaces (or scopes) for labels and identifiers. For example, in a function declaration:

```
function Com( R : byte) : byte;

var
  B : char;
  K : byte;

begin
  ...
end;
```

first line of the declaration is the function heading and all of the succeeding lines make up the block. B and K are local variables; their declarations apply only to the Com function block and override - in this block only - any declarations of the same identifiers that may occur in the program block or implementation section of a unit.

PROGRAM ORGANIZATION

Program elements (constants, variables and routines) need to be declared in their proper place in the code. Otherwise, compiler may not be able to comprehend the program correctly.

Organization of the main unit should have the following form:

```
program program_name;       // program name
uses ...                    // include other units

const ...                   // constants declaration
var ...                     // variables declaration

procedure procedure_name    // procedures declaration
  begin
    ...
  end;

function function_name      // functions declaration
  begin
    ...
  end;

begin                       // beginning of program

  ...                       // program body

end.                        // end of program
```

Organization of other units should have the following form:

```
unit unit_name;              // unit name
uses ...                     // include other units
implementation               // must follow the keyword uses

const ...                    // constants declaration
var ...                      // variables declaration

procedure procedure_name     // procedures declaration
  begin
    ...
  end;

function function_name       // functions declaration
  begin
    ...
  end;

  ...                        // unit body

end.                         // end of unit
```

TYPE CONVERSION

mikroPascal is capable of both implicit and explicit conversion of data types. Implicit conversion is the one automatically performed by compiler. On the other hand, explicit conversion is performed only on demand issued by user.

This means that you can, obeying a few rules, combine simple data types with any operators to create legal expressions and statements. Refer to data types if you are not familiar with data types supported by mikroPascal.

As stated in the chapter about operators, you cannot mix signed and unsigned data types in expressions that contain arithmetic or logical operators. You can assign signed to unsigned or vice versa only using the explicit conversion.

Implicit Conversion

- Implicit conversion takes place between byte and word, so you can combine byte and word with any operators to form legal expressions.
- Implicit conversion takes place between short, integer and longint so you can combine short, integer and longint with any operators to form legal expressions.
- Relation operators can be used without any restraints. Smart algorithm governing relation operators allows comparing any two data types.
- The compiler provides automatic type conversion when an assignment is performed, but does not allow to assign signed data type to unsigned and vice versa.

You can find more information on implicit conversion in chapters Assignment and Implicit Conversion, and Implicit Conversion and Legal Expressions.

Explicit Conversion

Explicit conversion can be executed at any point by inserting type (byte, word, short, integer, or longint) ahead of the expression to be converted. The expression must be enclosed in parentheses. You can't execute explicit conversion on the operand left of the assignment operator.

Special case is conversion between signed and unsigned. It is important to understand that explicit conversion between signed and unsigned data does not change binary representation of data; it merely allows copying of source to destination.

Example 1:

```
var tA, tB, tC : byte;

if tA + tB > tC then tA := 0;
```

This could be wrong, because there is an expression on the left. Compiler evaluates it, and treats it as a variable of type that matches type of tA or tB (the larger of the two); in this case - a byte.

```
tA := 250;
tB := 10;
tC := 20;

if tA + tB > tC then tA := 0;
```

In this case, since the result of the expression is treated as byte, we get that 250 + 10 is lower than 20. Actually, the result of the expression is truncated to byte: 250 + 10 is 4, and 4 is lower than 20.

But if we wrote it like this:

```
if word(tA + tB) > tC then tA := 0;
```

it would be correct, because we have explicitly instructed the compiler to treat tA + tB as a word. Hence, the result will equal 260 and greater than 20, returning the expected result.

Example 2:

Explicit conversion can also be used when you are sure which type you want to convert expression to. Consider the following lines:

```
var tA, tB, tC : byte;
    A_, B_: short

tA := byte(A_);
B_ := short(tA + tB * tC);
```

It is important to understand that explicit conversion between signed and unsigned data does not change binary representation of data; it only allows copying source to destination. Thus, if A_ was -1, its binary representation would be 11111111, and A would become 255.

Even if you have ordered the explicit conversion, compiler will perform implicit if necessary.

Example 3:

You cannot execute explicit conversion on the operand left of the assignment operator:

```
word(b) := Bb; // compiler will report an error.
```

ASSIGNMENT AND IMPLICIT CONVERSION

Overview

mikroPascal provides automatic type conversion every time an assignment is performed. But it does not allow assigning signed data to unsigned and vice versa, because there is a significant risk of losing information.

Implicit conversion takes place when assignment is performed:

between byte and word
between short, integer, and longint

Notes

Destination will store the correct value only if it can properly represent the result of expression (that is, if the result fits in destination range).

Feel free to use operands of any size under the defined rules, but keep in mind that the PIC is optimized to work with bytes. Every operation involving more complex data types (word, integer or longint) will take more run time and more memory. So for the best possible results, use as small destinations and operands as you can.

Examples

```
A := B;
```

If A and B are of the same type, value of B is simply assigned to A. More interesting case is if A and B are of different types:

```
A : byte;
B : word;
...
B := $ff0f;
A := B;  // A becomes $0f, higher byte $ff is lost
```

If A is more complex than B, then B is extended to fit the correct result:

```
A : word;
B : byte;
...
B := $ff;
A := B; // A becomes $00ff
```

For signed types :

```
B_ : integer;
A_ : short;
...
A_ : = -10;
B_ : = A_; // B becomes -10
```

In hex representation, this means that the higher byte is sign extended.

```
C := expression;
```

Calculated value of the expression will be assigned to the destination. Part of the information may be lost if the destination cannot properly represent the result of the expression (i.e. if result can't fit in range of destination data type). Browse through examples for more details.

For example (this is correct):

```
C := A + B;
```

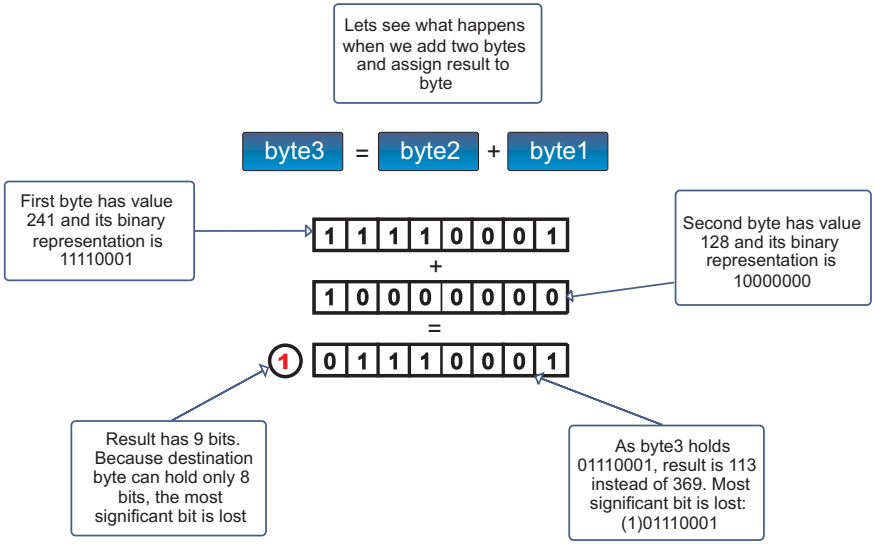
C is byte, so its range is 0 .. 255. If (A + B) fits in this range you will get the correct value in C.

```
A := 123;
B := 90;

C := A + B; // C becomes 213
```

But what happens when A + B exceeds the destination range? Let's assume the following:

```
A := 241;
B := 128;

C := A + B; // C becomes 113, obviously incorrect
```

See the following figure for closer explanation.


Lets see what happens when we add two bytes and assign result to byte
byte3 = byte2 + byte1
First byte has value 241 and its binary representation is 11110001
1 1 1 1 0 0 0 1
+
1 0 0 0 0 0 0 0
Second byte has value 128 and its binary representation is 10000000
=
1
0 1 1 1 0 0 0 1
Result has 9 bits. Because destination byte can hold only 8 bits, the most significant bit is lost
As byte3 holds 01110001, result is 113 instead of 369. Most significant bit is lost: (1)01110001

In order to fully understand this, we should recollect the data types.

Data type determines not only the range of values variable can hold, but also the amount of RAM space it consumes. This is fundamental in practical programming.

Let's assume that our destination variable C is a byte, consuming 8 bits of PIC RAM, spanning values 0 to 255. Now observe what really happens inside the PIC: the result should be 369, but in binary representation it equals (1)01110001. Because C is limited to 8 bits it will store the lower 8 bits while dropping the rest of the information (the most significant bit). 01110001 equals 113 in decimal representation.

```
var testA, testB: byte;
    Cc : word;

begin

  testA := 250;
  testB := 10;

  Cc := testA + testB; { this will always be correct because
            range for Cc is 0..65535 and maximum result
            of adding two bytes is only 255 + 255 = 510 }

end.
```

As already stated, destination will store the correct value only if it can properly represent the result of the expression (that is, the result fits in destination range).

```
var testA, testB: byte;
    Cc : word;
    Sa, Sb, Sc : short;
    Saa : integer;
    Sbbbb : longint;

begin
  testA := 250;
  testB := 10;
  Cc := testA * testB + testB; // Cc becomes 2510;
  Sb := 120;
  Sc := -100;
  Sa := Sb + Sc;  // Sa  becomes  20;
  Sa := Sb - Sc;  // Sa is short with range -127..128,
                  // thus, instead of 220,
                  // Sa becomes -36, because only
                  // lower 8 bits are registered;

  Saa := (Sb * Sc) div 13;
                  // Saa becomes -923;

  Sbbbb := integer(Sb * Sc) * Sc;
                  // Sbbbb becomes 1200000;
end.
```

IMPLICIT CONVERSION AND LEGAL EXPRESSIONS

Overview

To create legal expressions, you can:

1. combine byte and word with any operators,

2. combine short, integer, and longint (note that longint does not employ *, div, mod) with any operators,

3. use Relation operators

```
expression1 (relation operator) expression2
```

Expression1 and expression2 can be any legal expressions. Be sure to understand how implicit conversion works with relation operators.

Implicit Conversion and Relation Operators

Comparing variable or constant to variable or constant always produces correct results.

Comparing expressions requires a little more attention.

```
expression1 (relation operator) expression2
```

Expressions can be any legal expressions created with arithmetical or logical operators. Every expression no matter how complex, can be decomposed to a number of simple expressions. Simple expression is expression composed of just one operator and its operands. Operator is logical or arithmetical. Examine the rules presented below.

Rules for Comparing Expressions

1. Complex expression is decomposed to a number of simple expressions, with respect to operator precedence and overriding parenthesis.

2. Simple expression is now treated in the following manner: if operands are of the same type, operation is performed, assuming that the result is of the same type.

3. If operands are not of the same type, then less complex operand (speaking in terms of data range) is extended:

- If one operand is byte and another is word, byte is converted in word.
- If one operand is short and another is integer, short is converted to integer.
- If one operand is short and another is longint, short is converted to longint.
- If one operand is integer and another is longint, integer is converted to longint.

4. After the first expression is decomposed to simpler ones, each of these simpler ones is evaluated abiding the rules presented here.

Illustration

Expression `a + b + c` is decomposed like this:

First evaluate `a + b` and get `(value of a + b)`

This gives us another simple expression
`(value of a + b) + c`

Let's assume a and b are bytes and c is word, with values:

```
a := 23;
b := 34;
c := 1000;
```

Compiler first calculates value of `a + b` and assumes that the result is byte:
`a + b` gives 57.

As c is of word type, result of adding `a + b` is casted to word and then added to c:
`57 + c` is 1057.

Signed and unsigned numbers cannot be combined using arithmetical and logical operators. Rules presented above are not valid when assigning expression result to variable.

```
r := expression;
```

Refer to chapter Assignment and Implicit Conversion for details.

Note

When adding operands of the same type and assigning value to third operand, incorrect value may be proceeded if the result exceeds range of declared data type. Similar rules apply to other arithmetical operators.

For example, if a and b are bytes, and cc is word:

```
a  := 56;
b  := 200;
cc := 1000;
```

a + b equals 1, because result type is assumed to be same as the operands' type (byte).

Added to cc, we get 1001, instead of the expected 1256.

Solution is to simply instruct the compiler to evaluate expression as you explicitly define. For example, you could explicitly cast the expression, like this:

```
word(a + b + c).
```

As result fits in word range, we get 1256 as expected.

For more details, refer to chapter Explicit Conversion.

Comparing variables and constants always produces the correct results regardless of the operands' type.

Example

```
if A + B > A then
```

First, compiler evaluates the expression on the left. During runtime, result is stored in a variable of type that matches the largest data type in the expression. In this case it is byte, as variables A and B are both bytes.

This is correct if the value does not exceed range 0..255, that is, if A + B is less than 255.

Let's assume Aa is of word type :

```
if Aa + B > A ...
```

First, compiler evaluates the expression on the left. The result value is treated as type that matches the largest data type in the expression. Since Aa is word and B is byte, our result will be treated as word type.

This is correct if the value does not exceed range 0..65535, i.e. if Aa + B is less than 65535.

```
{ if C is less than zero C := (-C) :}

if  C < 0  then
  C := (-C);

{ Stay in loop while C is not equal to variable
  compare_match; increment C in every cycle: }

while C <> compare_match
 begin
    C := C + 1;
 end;
```

OPERATORS

There are three types of operators in mikroPascal:

Arithmetic Operators
Logical (Bitwise) Operators
Relation Operators (Comparison Operators)

Operator Precedence

Operator	Priority
not	first (highest)
*, div, mod, and, shl, shr	second
+, -, or, xor	third
=, <>, <, >, <=, >=	fourth (lowest)

In complex expressions, operators with higher precedence are evaluated before the operators with lower precedence; operators of equal precedence are evaluated according to their position in the expression starting from the left.

Example 1:

```
B and T + A
{ (bitwise and) B and T, then add A to the result;
  and is performed first, because it has precedence over + .}
```

Example 2:

```
A - B + D
{ first subtract B from A, then add D to the result;
  - and + have the equal precedence, thus the operation on
    the left is performed first. }
```

Example 3:

```
{ You can use parentheses to override these precedence rules.
  An expression within  parentheses is evaluated first, then
  treated as a single operand. For example: }

(A + B) * D
{ multiply D and the sum of A and B. }

A + B * D
{ first multiply B and D and then add A to the product. }
```

Rules for Creating Legal Expressions

1. You cannot mix signed and unsigned data types in expressions with arithmetic or logical operators. If you need to combine signed with unsigned, you will have to use explicit conversion.
   Example:

```
var Sa : short;
    teA : byte;
    Bb : word;
    Sbb : integer;
    Scccc : longint;
...
teA := teA + Sa;     // compiler will report an error
teA := teA and Sa;   // compiler will report an error

// But you can freely mix byte and word . .
    Bb := Bb + (teA * teA);

// . . and short with integer and longint;
    Scccc := Sbb * Sa + Scccc;
```

2. You can assign signed to unsigned, or unsigned to signed only using the explicit conversion. More details can be found in chapter Implicit Conversion and Assignment Operator.

```
Sa := short(teA);
// this can be done

Sa := teA
// this can't be done, compiler will report an error
```

3. Relation operators can be used with all data types, regardless of the sign.
   Example:

```
if Sa > teA then Sa := 0;
```

Notes for Relation Operators

Comparing variable or constant to variable or constant will always produce correct results.

Comparing expressions requires a little more attention - when compiler is calculating value of the expression to be compared, it first has to evaluate the expression. If the result of the expression exceeds the range of the largest data type in the expression, comparison will most likely be inaccurate. This can be avoided by using the explicit conversion.

More details can be found in chapter Implicit Conversion and Relation Operators.

Runtime Behavior

PIC MCUs are optimized for working with bytes. It takes less time to add two bytes than to add two words, naturally, and similar pattern is followed by all the other operators. It is a good practice to use byte or short data type whenever appropriate. Although the improvement may seem insignificant, it could prove valuable for applications which impose execution within time boundaries.

This is a recommendation which shouldn't be followed too literally - word, integer and longint are indispensable in certain situations.

Arithmetic Operators

Overview

Operator	Operation	Operand Types	Result Type
+	addition	byte, short, integer, words, longint	byte, short, integer, words, longint
-	subtraction	byte, short, integer, words, longint	byte, short, integer, words, longint
*	multiplication	byte, short, integer, words	integer, words, longint
div	division	byte, short, integer, words	byte, short, integer, words
mod	remainder	byte, short, integer, words	byte, short, integer, words

Mod and Div

`A div B` is the value of A divided by B rounded down to the nearest integer. The **mod** operator returns the remainder obtained by dividing its operands. In other words,

```
X mod Y = X - (X div Y) * Y.
```

If 0 (zero) is used explicitly as the second operand (i.e. X div 0), compiler will report an error and will not generate code. But in case of implicit division by zero: `X div Y`, where Y is 0 (zero), result will be the maximum value for the appropriate type (for example, if X and Y are words, the result will be $ffff).

Important

Destination will store the correct value only if it can properly represent the result of the expression (that is, if result fits in the destination range). More details can be found in chapter Assignment and Implicit Conversion.

Arithmetics and Data Types

mikroPascal is more flexible compared to standard Pascal as it allows both implicit and explicit type conversion. In mikroPascal, operator can take operands of different type; refer to chapter Type Conversion for more details. You cannot combine signed and unsigned data types in expressions with arithmetic operators.

Unsigned and Conversion

If number is converted from less complex to more complex data type, upper bytes are filled with zeros. If number is converted from more complex to less complex data type, data is simply truncated (upper bytes are lost).

Signed and Conversion

If number is converted from less complex to more complex data type, upper bytes are filled with ones if sign bit equals 1 (number is negative). Upper bytes are filled with zeros if sign bit equals 0 (number is positive). If number is converted from more complex to less complex data type, data is simply truncated (upper bytes are lost).

Example

```
program extr;

var Sa : short;
     A : byte;
    Bb : word;
   Sbb : integer;
 Scccc : longint;

begin

A := A + Sa;
// compiler will report an error,
// mixing signed with unsigned;

A := A - Sa;
// compiler will report an error,
// mixing signed with unsigned;

// But you can freely combine byte with word . .

Bb := Bb + (A * A);

// . . and short with integer and longint;

Scccc := Sbb * Sa + Scccc;

end.
```

Unary arithmetic operators

Operator	Operation	Operand Types	Result Types
+(unary)	sign identity	short, integer, longint	short, integer, longint
- (unary)	sign negation	short, integer, longint	short, integer, longint

Unary arithmetic operators can be used to change sign of variables:

```
a := 3;
b := -a;
// assign value -3 to b
```

Runtime Behavior

PIC microcontrollers are optimized to work with bytes. Refer to PIC MCU Specifics.

Boolean Operators

Boolean operators are not true operators, because there is no boolean data type defined in mikroPascal.

These "operators" conform to standard Boolean logic. They cannot be used with any data type, but only to build complex conditional expression.

Operator	Operation
not	negation
and	conjunction
or	disjunction

Example

```
if (astr > 10) and (astr < 20) then
PORTB:= $ff;
```

Logical (Bitwise) Operators

Overview

Operator	Operation	Operand Types	Result Types
not	bitwise negation	byte, word, short, integer, long	byte, word, short, integer, long
and	bitwise conjunction	byte, word, short, integer, long	byte, word, short, integer, long
or	bitwise disjunction	byte, word, short, integer, long	byte, word, short, integer, long
xor	bitwise xor	byte, word, short, integer, long	byte, word, short, integer, long
shl	bit shift left	byte, word, short, integer, long	byte, word, short, integer, long
shr	bit shift right	byte, word, short, integer, long	byte, word, short, integer, long

SHR and SHL

shl : shift left the operand for a number of bit places specified in the right operand (must be positive and less then 255).

shr : shift right the operand for a number of bit places specified in the right operand (must be positive and less then 255).

For example, if you need to extract the higher byte, you can do it like this:

```
var temp : word;

begin
  TRISA := word(temp shr 8);
end.
```

Important

Destination will hold the correct value if it can properly represent the result of the expression (that is, if result fits in destination range). More details can be found in chapters Type Conversions and Assignment and implicit Conversion.

Logical Operators and Data Types

mikroPascal is more flexible compared to standard Pascal as it allows both implicit and explicit type conversion. Note that you cannot mix signed and unsigned data types in expressions with logical operators.

Unsigned and Conversion

If number is converted from less complex to more complex data type, upper byte is filled with zeros;

If number is converted from more complex to less complex data type, data is simply truncated (upper bytes are lost).

Example for unsigned and logical operators :

```
var  teA : byte;
     Bb : word;

begin
  Bb := $F0F0;
  teA := $aa;
  Bb := Bb and teA;  // Bb becomes $00a0;
end.
```

In this case, teA is treated as a word with upper byte equal to 0 prior to the operation.

Signed and Conversion

If number is converted from less complex data type to more complex, upper bytes are filled with ones if sign bit is 1 (number is negative); upper bytes are filled with zeros if sign bit is 0 (number is positive).

If number is converted from more complex data type to less complex, data is simply truncated (upper bytes are lost).

```
var  Sa : short;
     Sbb : integer;

begin
  Sbb := $70FF;
  Sa := -12;
  Sbb := Sbb and Sa;  // Sbb becomes $70f4;
end.
```

In this case, Sa is treated as an integer with the upper byte equal to $FF (this in fact is sign extending of short to integer) prior to the operation.

```
begin
  Sbb := $OFF0;
  Saa := $0a;
  Sbb := Sbb and Sa;  // Bb becomes $0000;
end;
```

In this case, Sa is treated as an integer with the upper byte equal to $00 (this in fact is sign extending of short to integer) prior to the operation.

Example

```
var teA, teB, teC : byte;
{ The logical operators perform bitwise manipulation
  on the operands. For example, if the value stored in
  teA (in binary) is 00001111 and the value stored in teB
  is 10000001, the following statements.. }

begin
teA := $0F;  { .. assign value 00001111 to teA. }
teB := $81;  { .. assign value 10000001 to teB. }

teC := teA or teB;
{ Performs bitwise or with teA, teB and the
  result is assigned to teC (value 10001111) }

teC := not teA;
{ Performs bitwise not with teA and the
  result is assigned to teC (value 11110000) }

teC := teA  shl  4;
{  shl shift to the left (operand on the left) for
   a number of positions specified in the operand on
   the right; operand on the right must be positive;
   in this example teC becomes $F0 }
// All bits shifted in are zeros.

teC := teA  shr  4;
{ shr shift to the right (operand on the left) for
  a number of positions specified in operand on
  the right; operand on the right must be positive;
  in this example C becomes $00 }

// New bits shifted in are zeros if operand type is
// byte/word sign extended for short, word, integer.
end.
```

```
{ You cannot mix signed and unsigned data types in
  expressions with logical operators: }

var   Sa : short;
      teA : byte;
      Bb : word;
      Sbb : integer;
      Scccc : longint;

begin
teA := teA + Sa;    // compiler will report an error;
teA := teA and Sa   // compiler will report an error;

// But you can freely mix byte with word . .
Bb := Bb and ( not teA);

// . . and short with integer and longint.
Scccc := Sbb xor Sa or Scccc;
end.
```

Runtime Behavior

PIC microcontrollers are optimized to work with bytes. Refer to PIC MCU Specific.

Relation (Comparison) Operators

Overview

Operator	Operation	Operand Types	Result Types
=	equality	All simple types	True or False
<>	inequality	All simple types	True or False
<	less-than	All simple types	True or False
>	greater-than	All simple types	True or False
<=	less-than-or-equal	All simple types	True or False
>=	greater-than-or-equal	All simple types	True or False

Relation operators (Comparison Operators) are commonly used in conditional and loop statements to control the program flow.

In general case:

Expression1 (relation operator) Expression2,

expression1 and expression2 can be any legal expression. Be familiar with how implicit conversion works with relations operators. You can compare signed and unsigned values.

Rules for Comparing Expressions

1. Complex expression is decomposed to a number of simple expressions, with respect to operator precedence and overriding parenthesis.

2. Simple expression is now treated in the following manner: if operands are of the same type, operation is performed, assuming that the result is of the same type.

3. If operands are not of the same type, then less complex operand (speaking in terms of data range) is extended:

If one operand is byte and another is word, byte is converted in word.

If one operand is short and another is integer, short is converted to integer.

If one operand is short and another is longint, short is converted to longint.

If one operand is integer and another is longint, integer is converted to longint.

4. After the first expression is decomposed to simpler ones, each of these is evaluated abiding the rules presented here.

Illustration

Expression a + b + c is decomposed like this:

First evaluate a + b and get (value of a + b)

This gives us another simple expression
(value of a + b) + c

Let's assume a and b are bytes and c is word,
with values:

```
a := 23;
b := 34;
c := 1000;
```

Compiler first calculates value of a + b and assumes
that the result is byte: a + b gives 57.

As c is of word type, result of adding a + b is
casted to word and then added to c: 57 + c is 1057.

Signed and unsigned numbers cannot be combined using arithmetical and logical operators. Rules presented above are not valid when assigning expression result to variable.

```
r := expression;
```

Refer to Assignment and Implicit Conversion for more details.

Examples

Comparing variables and constants always produces the correct results regardless of the operands' type.

```
if A > B then ...;
if A > 47 then ...;

if A + B > A ...
```

First, compiler evaluates the expression on the left. During the run-time, result is stored in a variable of type that matches the largest data type in the expression. In this case it is byte, as variables A and B are both bytes.

This is correct if the value does not exceed range 0..255, that is, if A + B is less then 255.

Let's assume Aa is of word type :

```
if Aa + B > A ...
```

First, compiler evaluates the expression on the left. The result value is treated as type that matches the largest data type in the expression. Since Aa is word and B is byte, our result will be treated as word type.

This is correct if the value does not exceed range 0..65535, that is, if A + B is less then 65535.

```
{ if C is less then zero C := (-C) : }

if  C < 0  then
  C := (-C);


{ Stay in loop while C is not equal to variable
  compare_match; increment C in every cycle: }

while C <> compare_match
 begin
    C := C + 1;
 end;
```

CONDITIONAL STATEMENTS

Conditional statements control which part(s) of the program will be executed, depending on a specified criteria. There are two conditional statements in mikroPascal:

IF statement
CASE statement.

We suggest browsing the chapters Relation Operators and Implicit Conversion and Relation Operators, if you have not done so already.

Labels and Goto

Labels represent a more clear-cut way of controlling the program flow. You can declare a label below variables declarations, but you cannot declare two labels under the same name within the same routine. Name of the label needs to be a valid identifier. Multiple label declarations in single line are not allowed.

Goto statement jumps to the specified label unconditionally, and the program execution continues normally from that point on.

Here is an example:

```
program test;

var jjj : byte;
label first;
label second;
label myLabel;

begin
 { some instructions... }
goto myLabel;
 { some instructions... }
myLabel:
 { some instructions... }
end.
```

Case Statement

Case statement is used for selecting one of several available branches in the program course. Case statement consists of a selector variable as a switch condition, and a list of possible values. These values can be constants, numerals, or expressions.

Eventually, there can be an else statement which is executed if none of the labels corresponds to the value of the selector.

Declaration of case statement is:

```
case selector of
  Values_1: statement_1;
  Values_2: statements_2;
  ...
  Valuse_N: statements_n;
end;
```

where *selector* is any variable of simple type or expression, and each *Values_x* is a comma-delimited sequence of expressions.

Case statement can have a final else clause:

```
case selector of
  Values_1: statements_1;
  Values_2: statements_2;
  ...
  Values_N: statements_N;
else
  statements;
end;
```

As soon as the case statement is executed, at most one of the statements *statement1 .. statementn* will be executed. The *caseValue* which matches the selector determines the statement to be used.

If none of the *Values* matches the *selector*, then the *statements* in the else clause (if there is one) are executed.

You can specify that more than one statement should be executed by enclosing those statements by begin and end. See the example below.

Example

```
case state
     0, state0, state1, state2 :  msg = 'OK';
     -1, state0 or errorFlag, state1 or errorFlag : msg = 'Error';
   else
        msg = 'No input';
end;

 { In case there are multiple matches, the first matching block
   will be executed.
   For example, if state2 equals -1, msg will be OK, not Error. }

...

case message_flag
     0:
        begin
          opmode = 0;
          LCD_Out(1, 1, 'Test Message 0');
        end;
     1, 2, 3, 4:
        begin
          opmode = 1;
          LCD_Out(1, 1, 'Test Message 1');
        end;
     5, 6, 7:
        begin
          opmode = 2;
          LCD_Out(1, 1, 'Test Message 2');
        end;
end;
```

If Statement

There are two forms of if statement:

Syntax of an if..then statement is:

```
if expression then statement
```

where *expression* returns a True or False value. If *expression* is True, then *statement* is executed, otherwise it's not.

Syntax of if..then..else statement is:

```
if expression then statement1 else statement2
```

where *expression* returns a True or False value. If *expression* is True, then *statement1* is executed; otherwise *statement2* is executed. *Statement1* and *statement2* can be statements of any type.

Note that there should never be a semicolon before the reserved word **else**.

Nested IF

Nested if statements require additional attention. General rule is that the nested conditionals are parsed starting from the innermost conditional, with each else bound to the nearest available if on its left.

```
if expression1 then if expression2
                 then statement1 else statement2;
```

Compiler treats the construction like this:

```
if expression1 then [ if expression2
                 then statement1 else statement2 ]
```

To force the compiler to interpret our example the other way around, we would have to write it explicitly:

```
if expression1 then
 begin
  if expression2 then
        statement1;
 end
else
  statement2;
```

Example

```
if J <> 0 then Res := I div J;

if j <> 0 then
begin
  i := i+1;
  j := 0;
end;

 ...

if v=0 then
 begin
    portb := por2;
    porta := 1;
    v := 1;
 end
else
 begin
   portb := por1;
   porta := 2;
   v := 0;
 end;
```

LOOPS

Loops are a specific way to control the program flow. By using loops, you can execute a sequence of statements repeatedly, with a control condition or variable to determine when the execution stops.

You can use the standard break and continue to control the flow of a repeat, while, or for statement. Break terminates the statement in which it occurs, while continue begins executing the next iteration of the sequence.

mikroPascal has three kinds of control loop instructions :

REPEAT statement
WHILE statement
FOR statement

Runtime Behavior

Note that certain operations may take longer time to be executed, which can lead to undesired consequences.

If you add two variables of short type and assign the result to short, it will be faster than to add two longint and assign value to longint, naturally.

Take a look at the following code :

```
var Sa, Sb : short;
    Saaaa, Sbbbb : longint;

begin
for Sa := 0 to 100 do
      Sb := Sb + 2;

for Saaaa := 0 to 100 do
      Sbbbb := Sbbbb + 2;
end.
```

PIC will execute the first loop *considerably* faster.

For Statement

For statement requires you to specify the number of iterations you want the loop to go through. Syntax of for statement is:

```
for counter := initialValue to finalValue do statement
```

or

```
for counter := initialValue downto finalValue do statement
```

where *counter* is variable; *initialValue* and *finalValue* are expressions compatible with *counter*; *statement* is any statement that does not change the value of *counter*.

Statement can be a compound statement:

```
for counter := initialValue downto finalValue do
begin
  statement1
  ...
  statementN;
end;
```

Every statement between **begin** and **end** will be executed once for each iteration.

Endless Loop

Be careful not to create endless loop by mistake. The following statement:

```
for counter := initialValue to finalValue do statement
```

will result in an an endless loop if *finalValue* is greater than, or equal to maximum value of *counter* data type. For example, this will be an endless loop, if *counter* is of byte type:

```
for counter := 0 to 255 do
  nop;
```

or

```
for counter := 0 to 500 do
  nop;
```

Also, the following statement:

```
for counter := initialValue downto finalValue do
  statement
```

will result in an endless loop if *finalValue* is less than or equal to minimum value of *counter* data type. For example, this is an endless loop, if *counter* is of byte or word type:

```
for counter := 234 downto 0 do
  nop;
```

Example

Here is a simple example of a for loop used for emitting hex code on PORTB. Nine digits will be printed with one second delay.

1. by incrementing the counter:

```
for i := 1 to 9  do
    begin
      portb := i;
      delay_ms(1000);
    end;
```

2. by decrementing the counter :

```
for i := 9 downto 1  do
    begin
      portb := i;
      delay_ms(1000);
    end;
```

Repeat Statement

Syntax of repeat statement is:

```
repeat
 begin
  statement_1;
  ...;
  statement_N;
 end;
until
expression;
```

where *expression* returns a True or False value. The repeat statement executes *statement_1; ...; statement_N* continually, checking the *expression* after each iteration. Eventually, when *expression* returns True, the repeat statement terminates.

The sequence is executed at least once because the check takes place in the end.

In mikroPascal, all statements between **repeat** and **until** must be enclosed within **begin** and **end**.

Example

```
i := 0;
repeat
  begin
    i := i + 1;    // execute these 2 statements repeatedly
    PORTB := i;    // until i equals 10 (ten);
  end;
until i = 10;
```

While Statement

Syntax of while statement is :

```
while expression do
begin
  statement_0;
  statement_1;
  ...
  statement_N;
end;
```

or like this, when you have only one statement to be executed:

```
while expression do statement_0;
```

Expression returns a True or False value and *statement* can be any kind of statement. All statements enclosed by **begin** and **end** are treated as one statement (compound statement).

Expression is tested first. If it returns True, all the following statements enclosed by **begin** and **end** will be executed (or only one statement, alternatively). It will keep on executing statements until the *expression* returns False.

Eventually, as *expression* returns False, while will be terminated without executing *statements*.

While is similar to Repeat, except the check is performed at the beginning of loop. If *expression* returns False upon first test, *statements* will not be executed.

Example

```
while I < 90 do
I := I + 1;

      ...

while I > 0 do
  begin
    I := I div 3;
    PORTA := I;
  end;
```

ASM Statement

Sometimes it can be useful to write part of the program in assembly. ASM statement can be used to embed PIC assembly instructions into Pascal code.

Note that you cannot use numerals as absolute addresses for SFR or GPR variables in assembly instructions. You may use symbolic names instead (listing will display these names as well as addresses). Also, you cannot use Labels in assembly; instead, you can use relative jumps such as goto $-1.

Declaration of asm statement is:

```
asm
  statementList
end;
```

where *statementList* is a sequence of assembly instructions.

Be careful when embedding assembly code - mikroPascal will not check if assembly instruction changed memory locations already used by Pascal variables.

Also, you cannot write comments in assembly.

Example

```
asm
  movlw 67
  movwf TMR0
end;
```

```
asm // second instruction is incorrect, see above
  MOVLW 0
  MOVWF $5
  MOVWF PORTA
end; // note that you cannot write comments in assembly
```

PIC MCU SPECIFIC

In order to get the most from your mikroPascal compiler, you should be familiar with certain aspects of PIC MCU. This chapter is not essential, but it can provide you a better understanding of PICs' capabilities and limitations, and their impact on the code writing.

For start, you should know that arithmetical operations such as addition and subtraction are carried out by ALU (Arithmetical Logical Unit). With PIC MCUs (series PIC16 and PIC18), ALU is optimized for working with bytes. mikroPascal is capable of handling much more complex data types, but note that these can increase the time needed for performing even simple operations.

Also, not all PIC MCU models are of equal performance. PIC16 series lacks hardware resources to multiply two bytes in HW - it is carried out by software algorithm generated by mikroPascal. On the other hand, PIC18 series has HW multiplier, and as a result, multiplication works considerably faster.

Loops are convincing examples of byte type efficiency, especially if statements repeated hundreds of times are involved. Consider the following lines:

```
for i := 1 to 100 do tA := ta + 1;
```

and

```
for ii := 1 to 100 do Aa := Aa + 1;
```

where *i* and *A* are variables of byte type,
and *ii* and *Aa* are variables of word type.

First loop will be executed *considerably* faster.

Although memory management is completely under the compiler's control, you can explicitly assign address to variable by means of directive absolute. See Directives for more information.

NOTE : Be aware that nested function and procedure calls have limited depth - 8 for PIC16 series and 31 for PIC18 series.

mikroPASCAL SPECIFIC

mikroPascal compiler was designed with reliability and comfort in mind. Thus, certain modifications of standard Pascal were necessary in order to make the compiler more PIC MCU compatible.

PIC SFR (Special Function Registers) are implicitly declared as global variables of byte type. Their scope is the entire project - they are visible in any part of the program or any unit. Memory management is completely under compiler's control, so there is no need to worry about PIC memory banks and storing the variables.

Interrupts can be easily handled in mikroPascal by means of predefined procedure interrupt. All you need to do is include the complete procedure definition in your program. mikroPascal saves the following SFR when entering interrupt: PIC12 and PIC16 series: W, STATUS, FSR, PCLATH; PIC18 series: FSR (fast context is used to save WREG, STATUS, BSR). Upon return from interrupt routine, these registers are restored.

NOTE: For PIC18 family, interrupts must be of high priority. mikroPascal does not support low priority interrupts.

For example, when handling the interrupts from TMR0 (if no other are allowed):

```
procedure interrupt;

begin
  counter := counter + 1;
  TMR0 := 96;
  INTCON := $20;
end;
```

In case of multiple interrupts enabled, you must test which of the interrupts occurred and then proceed with the appropriate code (interrupt handling):

```
procedure interrupt;

begin
 if TestBit(INTCON,TMR0IF)=1 then
    begin
      counter := counter + 1;
      TMR0 := 96;
      ClearBit(INTCON,TMR0F);
    end

    else
      if TestBit(INTCON,RBIF)=1 then
        begin
          counter := counter + 1;
          TMR0 := 96;
          ClearBit(INTCON,RBIF);
        end;
end;
```

See also:
Built-in Functions and Procedures
Library Functions and Procedures

COMPILER ERROR MESSAGES

Error Messages

Type of Error	Error No.
_SYNTAX_ERROR	100
_NOT_VALID_IDENT	101
_INVALID_STATEMENT	102
_STACK_OVERFLOW	103
_INVALID_OPERATOR	104
_IF_ELSE_ERROR	105
_VARIABLE_EXPECTED	106
_CONSTANT_EXPECTED	107
_ASSIGNMENT_EXPECTED	108
_BREAK_ERROR	109
_UNKNOWN_TYPE	110
_VARIABLE_REDECLARED	111
_VARIABLE_NOT_DECLARED	112
_MAX_LINE_NUMBER_EXCEEDED	113
_ALREADY_DECLARED // *for proc and func*	114
_TOO_MANY_PARAMS	115
_NOT_ENOUGH_PARAMS	116
_TYPE_MISMATCH	117
_FILE_NOT_FOUND	118
_NOT_ENOUGH_RAM	119
_USES_IN_BETA_V	120
_INTERNAL_ERROR	121
_NOT_ENOUGH_ROM	122
_INVALID_ARRAY_TYPE	123
_BAUD_TOO_HIGH	124
_DIVISION_BY_ZERO	125
_INCOMPATIBLE_TYPES	126
_TOO_MANY_CHARACTERS	127
_OUT_OF_RANGE	128
_USES_POSITION	129
_INVALID_ASM_COMMAND	130
_OPERATOR_NOT_APPLICABLE	131
_EXPRESSION_BY_ADDRESS	132
_IDENTIFIER_EXPECTED	133
_MOVING_ARRAYS	134

Warning Messages

Type of Error	Error No.
_CODE_AFTER_END	200
_BAUD_ERROR	201
_UPPER_BYTES_IGNORED	202
_UPPER_WORDS_IGNORED	203
_IMPLICIT_TYPECAST	204

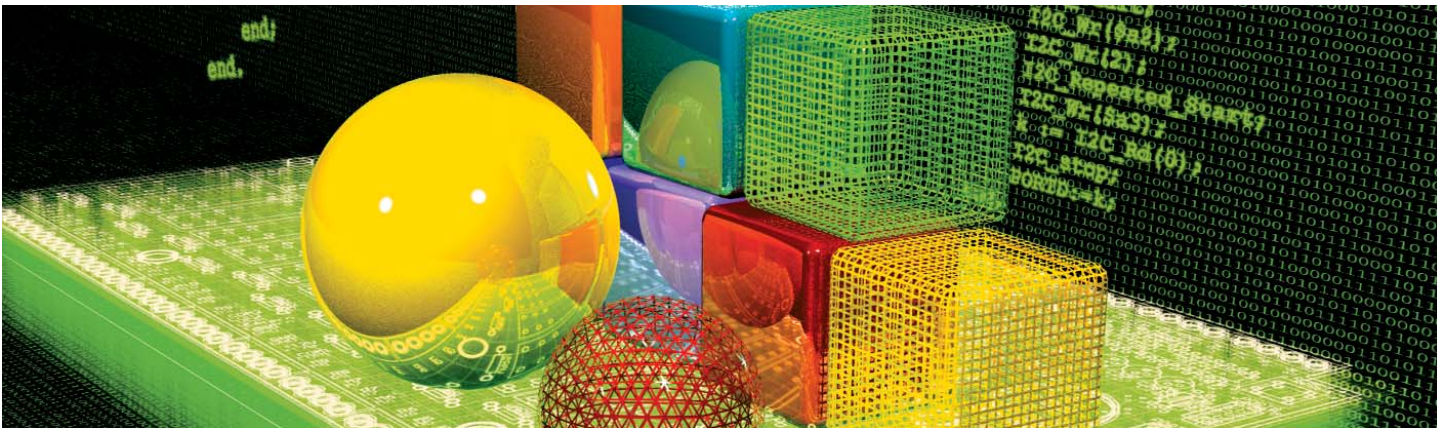
Hint Messages

Type of Error	Error No.
_VAR_NOT_USED	300
_PROC_NOT_CALLED	301

Linker Error Messages

Type of Error	Error No.
_UNKNOWN_ASM	400
_ADDRESS_CALC_ERROR	401

CHAPTER 3



Built-in and Library Routines

mikroPascal provides a number of built-in and library routines which help you develop your application faster and easier. Libraries for ADC, CAN, USART, SPI, I2C, 1-Wire, LCD, PWM, RS485, numeric formatting, bit manipulation, and many other are included along with practical, ready-to-use code examples.

BUILT-IN ROUTINES

mikroPascal compiler incorporates a set of built-in functions and procedures. They are provided to make writing programs faster and easier. You can call built-in functions and procedures in any part of the program.

Routines

```
procedure SetBit(var REG : byte; BIT : byte);
procedure ClearBit(var REG : byte; BIT : byte);
function  TestBit(var REG : byte; BIT : byte) : byte;

function  Lo( arg : byte..longint ) : byte;
function  Hi( arg : word..longint ) : byte;
function  Higher( arg : longint ) : byte;
function  Highest( arg : longint ) : byte;

procedure Inc(var arg : byte..longint);
procedure Dec(var arg : byte..longint);

procedure Delay_us(const COUNT : word);
procedure Delay_ms(const COUNT : word);
procedure Delay_Cyc(Cycles_div_by_10: byte);

function  Length(text : string ) : byte;
```

Routines SetBit, ClearBit and TestBit are used for bit manipulation. Any SFR (Special Function Register) or variable of byte type can pass as valid variable parameter, but constants should be in range [0..7].

Routines Lo, Hi, Higher and Highest extract one byte from the specified parameter. Check the examples for details.

Routines Inc and Dec increment and decrement their argument respectively.

Routines Delay_us and Delay_ms create a software delay in duration of COUNT microseconds or milliseconds, respectively.

Routine Delay_Cyc creates a delay based on MCU clock. Delay lasts for (10 times the input parameter) in MCU cycles. Input parameter needs to be in range 3 .. 255.

Function Length returns string length.

Examples

```
SetBit(PORTB,2);          // set PORTB bit RB2 to value 1

ClearBit(PORTC,7);        // clear PORTC bit RC7

TestBit(PORTA,2);         // returns 1 if PORTA bit RA2 is
                          // 1, and 0 if RA2 is 0

Lo(A);                    // returns lower byte of variable A
                          // ( byte 0, assuming that word/integer
                          // comprises bytes 1 and 0,
                          // and longint comprises bytes 3, 2, 1,
                          // and 0 )

Hi(Aa);                   // returns higher byte of variable Aa
                          // ( byte 1, assuming that word/integer
                          // comprises bytes 1 and 0,
                          // and longint comprises bytes 3, 2, 1,
                          // and 0 )

Higher(Aaaa);             // returns byte next to the highest byte
                          // of variable Aaaa ( byte 2, assuming
                          // that longint comprises bytes 3, 2, 1,
                          // 0 )

Highest(Aaaa);            // returns the highest byte of variable
                          // Aaaa ( byte 3, assuming that longint
                          // comprises bytes 3, 2, 1, 0 )

Inc(Aaaa);                // same as Aaaa := Aaaa + 1

Dec(Aaaa);                // same as Aaaa := Aaaa - 1


Delay_us(100);            // creates software delay equal to 100
                          // microseconds

Delay_ms(1000);           // creates software delay equal to 1000
                          // milliseconds = 1s

Delay_Cyc(100);           // creates delay of 1000 MCU cycles

Length(Text);             // returns string length as byte
```

LIBRARY ROUTINES

Library procedures and functions represent a set of routines. This collection of functions and procedures is provided for simplifying the initialization and use of PIC MCU and its hardware modules (ADC, I2C, USART, SPI, PWM), driver for LCD, drivers for internal and external CAN modules, flexible 485 protocol, numeric formatting routines...

Currently included libraries:

1wire
ADC
CAN
CANSPI
Compact Flash
Flash Memory
EEPROM
I2C
LCD (4-bit interface)
LCD8 (8-bit interface)
Graphic LCD
PWM
RS485
SPI
USART

Software I2C
Software SPI
Software UART

Sound
Manchester Code
Numeric Formatting Routines
Utilities

1-Wire Library

1-wire library provides routines for communicating via 1-wire bus, for example with DS1820 digital thermometer.

Note that oscillator frequency Fosc needs to be at least 4MHz in order to use the routines with Dallas digital thermometers.

Routines

```
function  OW_Reset(var PORT : byte; PIN : byte): byte;
function  OW_Read(var PORT : byte; PIN : byte): byte;
procedure OW_Write(var PORT : byte; PIN, par : byte);
```

```
function OW_Reset(var PORT : byte; PIN : byte): byte;
```

Issues 1-wire reset signal for DS1820.
Parameters PORT and pin specify the location of DS1820; return value of the function is 0 if DS1820 is present, and 1 if it is not present.

```
function OW_Read(var PORT : byte; PIN : byte): byte;
```

Reads one byte via 1-wire bus.

```
procedure OW_Write(var PORT : byte; PIN, par : byte);
```

Writes one byte (parameter par) via 1-wire bus.

Example

The following code demonstrates use of 1-wire library procedures and functions. The example reads the temperature using DS1820 connected to PORTA, pin 5. Be sure to set the Fosc appropriately in your project.

```
program onewire_test;

var i, j1, j2, por1, por2: byte;
    text : array[20] of char;

begin
   text    := 'Temperature:';
   PORTB   := 0;                  // initialize PORTB to 0
   PORTA   := 255;                // initialize PORTA to 255
   TRISB   := 0;                  // PORTB is output
   TRISA   := 255;                // PORTA is input
  LCD_Init(PORTB);
  LCD_Cmd(LCD_CURSOR_OFF);
  LCD_Out(1, 1, text);
 repeat
    begin
      OW_Reset(PORTA,5);          // 1-wire reset signal
      OW_Write(PORTA,5,$CC);      // issue command to DS1820
      OW_Write(PORTA,5,$44);      // issue command to DS1820
      Delay_us(120);
      i := OW_Reset(PORTA,5);
      OW_Write(PORTA,5,$CC);      // issue command to DS1820
      OW_Write(PORTA,5,$BE);      // issue command to DS1820
      Delay_ms(1000);
      j1 := OW_Read(PORTA,5);     // get result
      j2 := OW_Read(PORTA,5);     // get result
      j1 := j1 shr 1;             // assuming the temp. >= 0C
      ByteToStr(j1, text);        // convert j1 to text
      LCD_Out(2, 8, text);        // print text
      LCD_Chr(2, 10, 223);        // 'degree' character
      LCD_Chr(2, 11,'C');
      Delay_ms(500);
    end;
  until false;                    // endless loop
end.
```

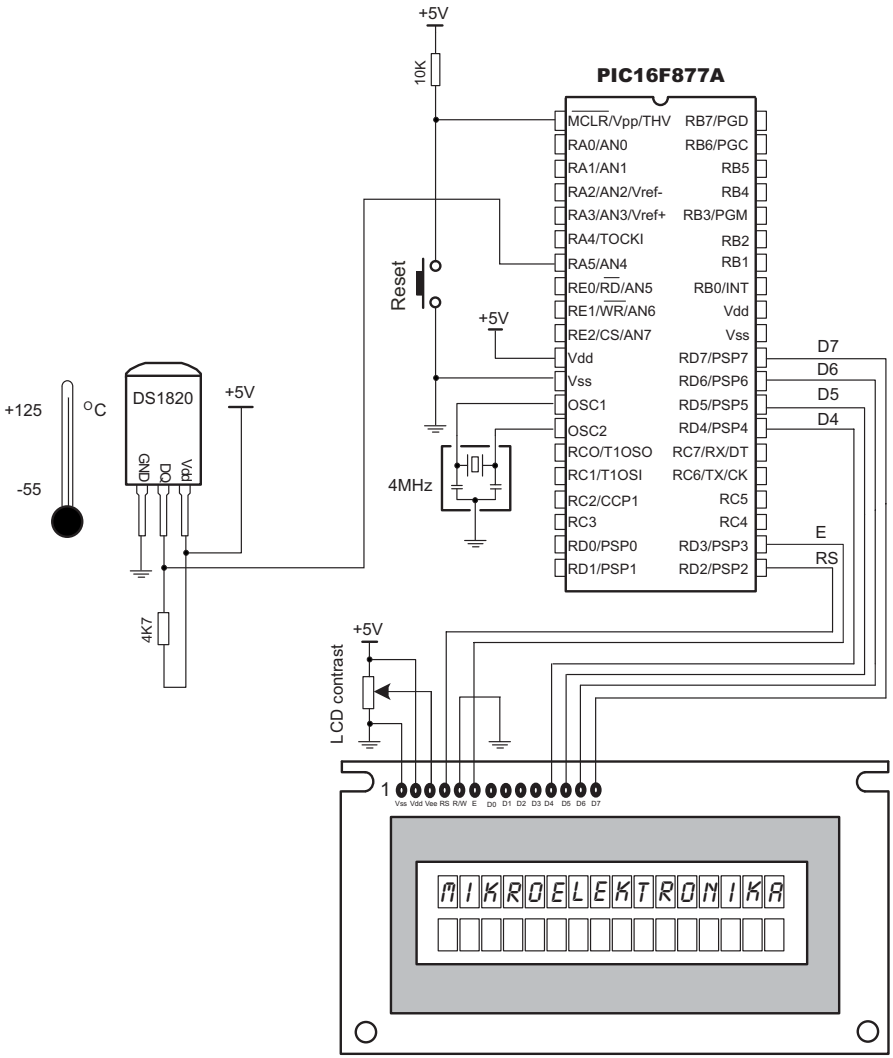


Figure (example of DS1820 on PORTA, pin 5)

ADC Library

ADC (Analog to Digital Converter) module is available with a number of PIC MCU models. Library function ADC_read is included to provide you comfortable work with the module.

The function is currently unsupported by the following PIC MCU models: P18F2331, P18F2431, P18F4331, and P18F4431.

Routines

You can use the library function to initialize internal AD converter, select channel, and get the result of conversion:

```
function ADC_Read(Channel : byte) : word;
```

It initializes ADC module to work with RC clock. Clock determines the time period necessary for performing AD conversion (min 12 Tad). RC sources typically have Tad 4uS (A/D conversion time per bit).

Parameter *Channel* determines which channel will be sampled. Refer to the device data sheet for information on device channels.

Important

Before using the function above, be sure to configure the appropriate TRISA bits to designate the pins as input. Also, configure the desired pin as analog input, and set Vref (voltage reference value).

Example

The following code demonstrates use of library function ADC_read. Example reads Channel 2 and stores value in variable temp_res.

```
program ADC_Test;
var temp_res : word;

begin
  ADCON1 := $80;    // configure analog inputs and Vref
  TRISA  := $ff;    // designate PORTA as input
  TRISB  := $3F;    // designate RB7,RB6 pins as outputs
  TRISD  := $0;     // designate PORTD as output
  while (true) do
  begin
    temp_res := ADC_read(2);
    PORTD := temp_res;    // send lower 8 bits to PORTD
    PORTB := word(temp_res shr 2);
      // send two most significant bits to PORTB
 end;
end.
```

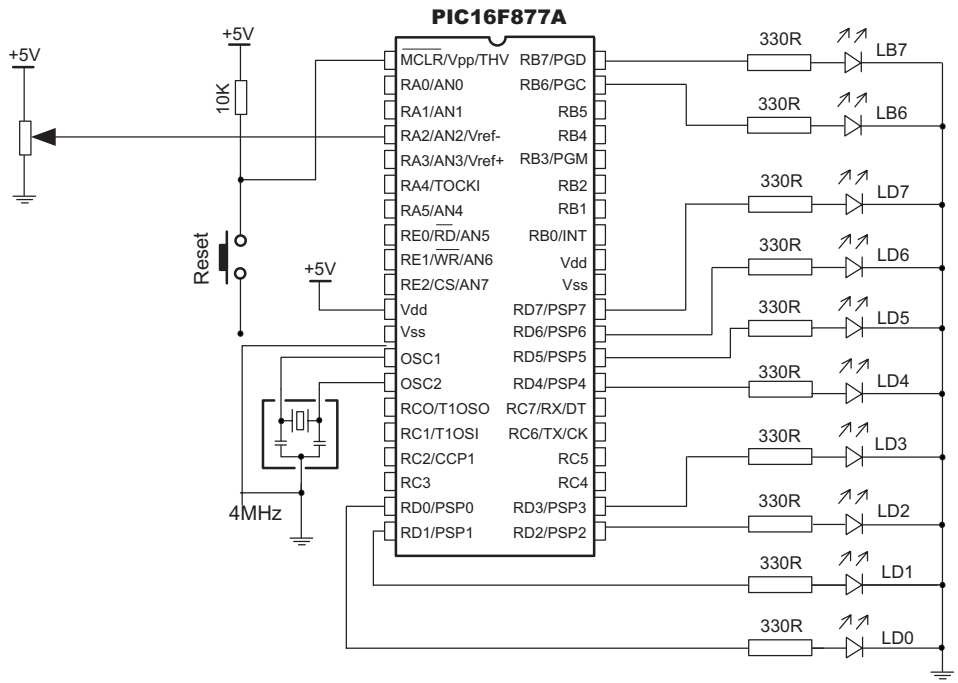


Figure (ADC HW connection)

CAN Library

CAN (Controller Area Network) module is available with a number of PIC MCU models. mikroPascal includes a set of library routines to provide you comfortable work with the module.

CAN routines are currently supported by PIC MCU models P18XXX8. Microcontroller must be connected to CAN tranceiver (MCP2551 or similar) which is connected to CAN bus.

The Controller Area Network module is a serial interface, useful for communicating with other peripherals or microcontrollers. Details about CAN can be found in appropriate literature and on mikroElektronika Web site.

Following routines can be considered a driver for CAN module on PIC MCUs.

```
procedure CANSetOperationMode(mode : byte, WAIT : byte);

function  CANGetOperationMode : byte;

procedure CANInitialize( SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS : byte);

procedure CANSetBaudRate( SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS : byte);

procedure CANSetMask( CAN_MASK : byte; val : longint; CAN_CONFIG_FLAGS : byte);

procedure CANSetFilter( CAN_FILTER : byte; val : longint;  CAN_CONFIG_FLAGS : byte);

function  RegsToCANID( var ptr : byte;  CAN_CONFIG_FLAGS : byte): longint;

procedure CANIDToRegs( var ptr: byte; val : longint;  CAN_CONFIG_FLAGS : byte);

function  CANwrite( id : longint; var Data : array[8] of byte;
                    DataLen, CAN_TX_MSG_FLAGS : byte) : byte;

function  CANread( var id : longint; var Data : array[8] of byte; var DataLen: byte;
                   var CAN_RX_MSG_FLAGS : byte) : byte;
```

CANSetOperationMode

Prototype: `procedure CANSetOperationMode(mode : byte; WAIT : byte);`

Parameters: mode - Operation mode code can take any of predefined constant values (see the constants below)
WAIT - Should have value TRUE(255) or FALSE(0)

Effects: CAN is set to requested mode

Overview: Given mode byte is copied to CANSTAT

Note: If WAIT is true, this is a blocking call. It won't return until requested mode is set.

If WAIT is false, this is a non-blocking call. It does not verify if CAN module is switched to requested mode or not. Caller must use CANGetOperationMode() to verify correct operation mode before performing mode specific operation.

CANGetOperationMode

Prototype: `function CANGetOperationMode : byte;`

Parameters: None

Output: Current operational mode of CAN module is returned

CANInitialize

Prototype:
```
procedure CANInitialize( SJW, BRP, PHSEG1, PHSEG2, PROPSEG,
CAN_CONFIG_FLAGS : byte);
```

Precondition: CAN must be in Configuration mode or else these values will be ignored.

Parameters:
SJW value as defined in 18XXX8 datasheet (must be between 1 thru 4)
BRP value as defined in 18XXX8 datasheet (must be between 1 thru 64)
PHSEG1 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PHSEG2 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PROPSEG value as defined in 18XXX8 datasheet (must be between 1 thru 8)
CAN_CONFIG_FLAGS value is formed from constants (see below)

Effects: CAN bit rate is set. All masks registers are set to '0' to allow all messages.

Filter registers are set according to flag value:

```
If (CAN_CONFIG_FLAGS and CAN_CONFIG_VALID_XTD_MSG) <> 0
    Set all filters to XTD_MSG
Else if (config and CONFIG_VALID_STD_MSG) <> 0
    Set all filters to STD_MSG
Else
    Set half of the filters to STD, and the rest to XTD_MSG.
```

Side Effects: All pending transmissions are aborted.

CANSetBaudRate

Prototype: `procedure CANSetBaudRate(SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS : byte);`

Precondition: CAN must be in Configuration mode or else these values will be ignored.

Parameters: SJW value as defined in 18XXX8 datasheet (must be between 1 thru 4)
BRP value as defined in 18XXX8 datasheet (must be between 1 thru 64)
PHSEG1 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PHSEG2 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PROPSEG value as defined in 18XXX8 datasheet (must be between 1 thru 8)
CAN_CONFIG_FLAGS - Value formed from constants (see section below)

Effects: CAN bit rate is set as per given values.

Overview: Given values are bit adjusted to fit in 18XXX8. BRGCONx registers and copied.

CANSetMask

Prototype: `procedure CANSetMask( CAN_MASK : byte; val : longint; CAN_CONFIG_FLAGS : byte);`

Precondition: CAN must be in Configuration mode. If not, all values will be ignored.

Parameters: CAN_MASK - One of predefined constant value
val - Actual mask register value.
CAN_CONFIG_FLAGS - Type of message to filter, either
CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG

Effects: Given value is bit adjusted to appropriate buffer mask registers.

CANSetFilter

Prototype:
```
procedure CANSetFilter(CAN_FILTER : byte; val : longint;
CAN_CONFIG_FLAGS : byte);
```

Precondition: CAN must be in Configuration mode. If not, all values will be ignored.

Parameters: CAN_FILTER - One of predefined constant values
val - Actual filter register value.
CAN_CONFIG_FLAGS - Type of message to filter, either
CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG

Effects: Given value is bit adjusted to appropriate buffer filter registers.

RegsTOCANID and CANIDToRegs

Prototypes:
```
function RegsToCANID(var ptr : byte; CAN_CONFIG_FLAGS : byte):
longint;

procedure CANIDToRegs(var ptr: byte; val : longint;
CAN_CONFIG_FLAGS : byte);
```

Effects: These two routines are used by other routines (internal purpose only)

CANWrite

Prototype:
```
function CANwrite( id : longint; var Data : array[8] of byte;
DataLen, CAN_TX_MSG_FLAGS : byte) : byte;
```

Precondition: CAN must be in Normal mode.

Parameters: id - CAN message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended).
Data - array of bytes up to 8 bytes in length
DataLen - Data length from 1 thru 8.
CAN_TX_MSG_FLAGS - Value formed from constants (see section below)

Effects: If at least one empty transmit buffer is found, given message is queued for the transmission. If none found, FALSE value is returned.

CANRead

Prototype:
```
function CANRead(var id : longint; var Data : array[8] of byte; var
DataLen : byte; var CAN_RX_MSG_FLAGS : byte) : byte;
```

Precondition: CAN must be in mode in which receiving is possible.

Parameters: id - CAN message identifier
Data - array of bytes up to 8 bytes in length
DataLen - Data length from 1 thru 8.
CAN_TX_MSG_FLAGS - Value formed from constants (see below)

Effects: If at least one full receive buffer is found, it is extracted and returned.
If none found, FALSE value is returned.

CAN Library Constants

You need to be familiar with constants that are provided for use with CAN library routines. See how to form values (from constants) that will be passed to or from routines in the example at the end of the chapter. All of the constants are predefined in CAN library.

CAN_OP_MODE

These constant values define CAN module operation mode. CANSetOperationMode() routine requires this code. These values must be used by itself, i.e. they cannot be ANDed to form multiple values.

```
const
 CAN_MODE_BITS : byte = $E0; // Use these to access opmode bits
 CAN_MODE_NORMAL : byte = 0;
 CAN_MODE_SLEEP  : byte = $20;
 CAN_MODE_LOOP   : byte = $40;
 CAN_MODE_LISTEN : byte = $60;
 CAN_MODE_CONFIG : byte = $80;
```

CAN_TX_MSG_FLAGS

These constant values define flags related to transmission of a CAN message. There could be more than one this flag ANDed together to form multiple flags.

```
const
 CAN_TX_PRIORITY_BITS : byte = $03;
 CAN_TX_PRIORITY_0 : byte = $FC;    // XXXXXX00
 CAN_TX_PRIORITY_1 : byte = $FD;    // XXXXXX01
 CAN_TX_PRIORITY_2 : byte = $FE;    // XXXXXX10
 CAN_TX_PRIORITY_3 : byte = $FF;    // XXXXXX11

 CAN_TX_FRAME_BIT : byte = $08;
 CAN_TX_STD_FRAME : byte = $FF;     // XXXXX1XX
 CAN_TX_XTD_FRAME : byte = $F7;     // XXXXX0XX

 CAN_TX_RTR_BIT : byte = $40;
 CAN_TX_NO_RTR_FRAME : byte = $FF; // X1XXXXXX
 CAN_TX_RTR_FRAME : byte = $BF;     // X0XXXXXX
```

CAN_RX_MSG_FLAGS

These constant values define flags related to reception of a CAN message. There could be more than one this flag ANDed together to form multiple flags. If a particular bit is set; corresponding meaning is TRUE or else it will be FALSE.

e.g.

```
if (MsgFlag and CAN_RX_OVERFLOW) <> 0

  // Receiver overflow has occurred.
  // We have lost our previous message.
  ...
```

```
const
 CAN_RX_FILTER_BITS : byte = $07; // For accessing filter bits
 CAN_RX_FILTER_1 : byte = $00;
 CAN_RX_FILTER_2 : byte = $01;
 CAN_RX_FILTER_3 : byte = $02;
 CAN_RX_FILTER_4 : byte = $03;
 CAN_RX_FILTER_5 : byte = $04;
 CAN_RX_FILTER_6 : byte = $05;
 CAN_RX_OVERFLOW : byte = $08;      // Set if Overflow else cleared
 CAN_RX_INVALID_MSG : byte = $10;   // Set if invalid else cleared
 CAN_RX_XTD_FRAME : byte = $20;     // Set if XTD msg else cleared
 CAN_RX_RTR_FRAME : byte = $40;     // Set if RTR msg else cleared
 CAN_RX_DBL_BUFFERED : byte = $80; // Set if this message was
                                    //  hardware double-buffered
```

CAN_MASK

These constant values define mask codes. Routine CANSetMask() requires this code as one of its arguments. These enumerations must be used by itself i.e. it cannot be ANDed to form multiple values.

```
const
 CAN_MASK_B1 : byte = 0;
 CAN_MASK_B2 : byte = 1;
```

CAN_FILTER

These constant values define filter codes. Routine CANSetFilter() requires this code as one of its arguments. These enumerations must be used by itself i.e. it cannot be ANDed to form multiple values.

```
const
 CAN_FILTER_B1_F1 : byte = 0;
 CAN_FILTER_B1_F2 : byte = 1;
 CAN_FILTER_B2_F1 : byte = 2;
 CAN_FILTER_B2_F2 : byte = 3;
 CAN_FILTER_B2_F3 : byte = 4;
 CAN_FILTER_B2_F4 : byte = 5;
```

CAN_CONFIG_FLAGS

These constant values define flags related to configuring CAN module. Routines CANInitialize() and CANSetBaudRate() use these codes. One or more these values may be ANDed to form multiple flags.

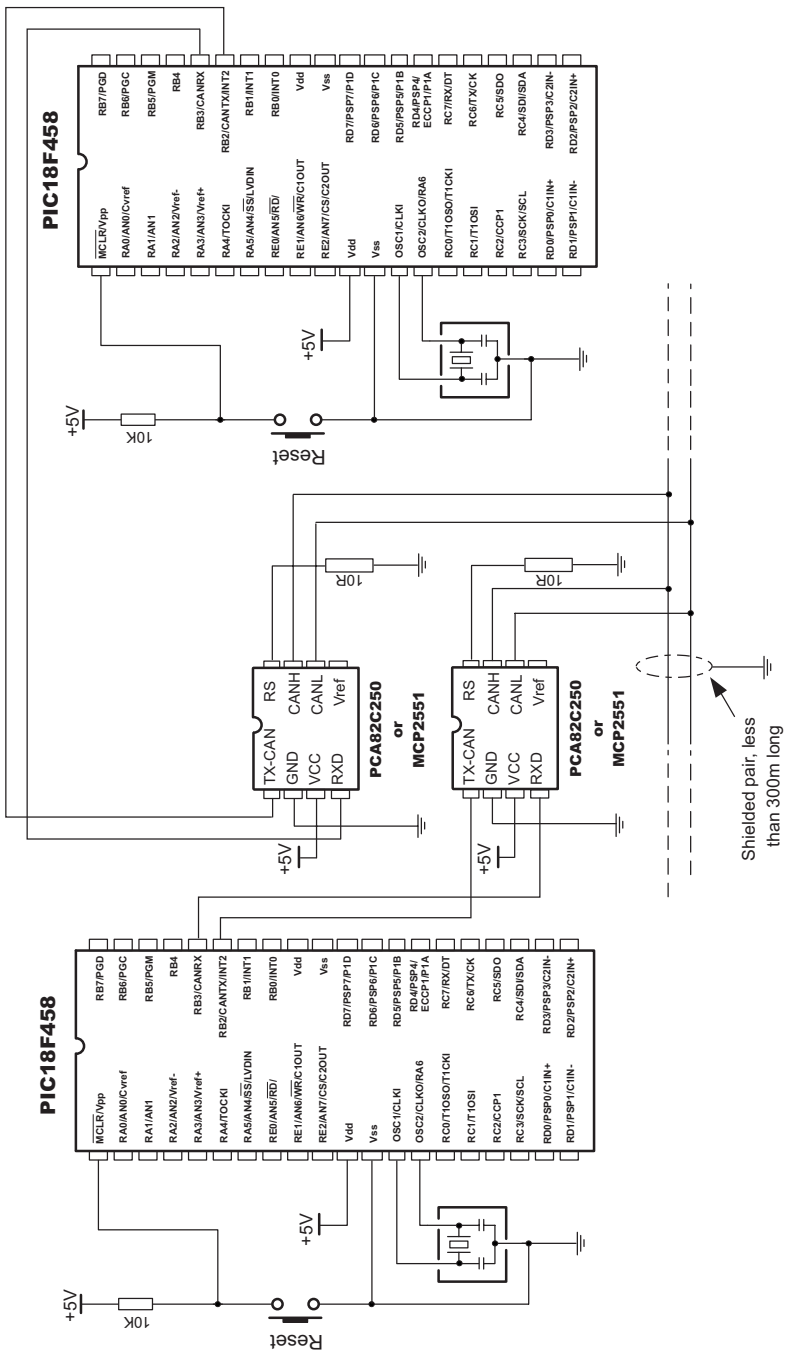
```
const
 CAN_CONFIG_DEFAULT : byte = $FF;              // 11111111
 CAN_CONFIG_PHSEG2_PRG_BIT : byte = $01;
 CAN_CONFIG_PHSEG2_PRG_ON : byte = $FF;        // XXXXXXX1
 CAN_CONFIG_PHSEG2_PRG_OFF : byte = $FE;       // XXXXXXX0

 CAN_CONFIG_LINE_FILTER_BIT : byte = $02;
 CAN_CONFIG_LINE_FILTER_ON : byte = $FF;       // XXXXXX1X
 CAN_CONFIG_LINE_FILTER_OFF : byte = $FD;      // XXXXXX0X

 CAN_CONFIG_SAMPLE_BIT : byte = $04;
 CAN_CONFIG_SAMPLE_ONCE : byte = $FF;          // XXXXX1XX
 CAN_CONFIG_SAMPLE_THRICE : byte = $FB;        // XXXXX0XX

 CAN_CONFIG_MSG_TYPE_BIT : byte = $08;
 CAN_CONFIG_STD_MSG : byte = $FF;              // XXXX1XXX
 CAN_CONFIG_XTD_MSG : byte = $F7;              // XXXX0XXX

 CAN_CONFIG_DBL_BUFFER_BIT : byte = $10;
 CAN_CONFIG_DBL_BUFFER_ON : byte = $FF;        // XXX1XXXX
 CAN_CONFIG_DBL_BUFFER_OFF : byte = $EF;       // XXX0XXXX
 CAN_CONFIG_MSG_BITS : byte = $60;
 CAN_CONFIG_ALL_MSG : byte = $FF;              // X11XXXXX
 CAN_CONFIG_VALID_XTD_MSG : byte = $DF;        // X10XXXXX
 CAN_CONFIG_VALID_STD_MSG : byte = $BF;        // X01XXXXX
 CAN_CONFIG_ALL_VALID_MSG : byte = $9F;        // X00XXXXX
```

Example This code demonstrates use of CAN library routines:

```
program CAN;

var aa,aa1,len,aa2 : byte;
    data : array[8] of byte;
    id : longint;
    zr, cont, oldstate: byte;

function TestTaster : byte;
begin
  result := true;
  if Button(PORTB, 0, 1, 0) then oldstate:= 255;
  if  oldstate and Button(PORTB, 0, 1, 1) then
  begin
      result := false;
      oldstate := 0;
  end;
end;


begin
  SetBit(TRISB,0);                    // designate pin RB0 as input
  PORTC := 0;
  TRISC := 0;
  PORTD := 0;
  TRISD := 0;
  aa := 0;
  aa1 := 0;
  aa2 := 0;

  aa1 :=  CAN_TX_PRIORITY_0 and     // form value to be used
          CAN_TX_XTD_FRAME and      //   with CANSendMessage
          CAN_TX_NO_RTR_FRAME;


  aa :=   CAN_CONFIG_SAMPLE_THRICE and   // form value to be used
          CAN_CONFIG_PHSEG2_PRG_ON and   //   with CANInitialize
          CAN_CONFIG_STD_MSG and
          CAN_CONFIG_DBL_BUFFER_ON and
          CAN_CONFIG_VALID_XTD_MSG and
          CAN_CONFIG_LINE_FILTER_OFF;


  { continues.. }
```

```
{ ..continued }

  cont := true;                  // upon signal change on RB0
  while cont do                  //   from logical 0 to 1
  begin                          //   proceed with program
    cont := TestTaster;          //   execution
  end;

  data[0] := 0;
  CANInitialize( 1,1,3,3,1,aa);  // initialize CAN
  CANSetOperationMode(CAN_MODE_CONFIG,TRUE); // set CONFIG mode

  ID := -1;
  CANSetMask(CAN_MASK_B1,ID,CAN_CONFIG_XTD_MSG); // mask1 to ones
  CANSetMask(CAN_MASK_B2,ID,CAN_CONFIG_XTD_MSG); // mask2 to ones
  CANSetFilter(CAN_FILTER_B1_F1,3,CAN_CONFIG_XTD_MSG);
    // set id of B1_F1 to 3

  CANSetOperationMode(CAN_MODE_NORMAL,TRUE); // set NORMAL mode
  portd := $FF;
  id := 12111;
  CANSendMessage(id,data,1,aa1); // send message via CAN

  while true do
  begin
    oldstate:=0;
    zr := CANReceiveMessage(id , Data , len, aa2);
    if (id = 3) and zr then
    begin
      portd := $AA;
      portc := data[0];              // output data at portC
      data[0] := data[0]+1;
      id := 12111;
      CANSendMessage(id,data,1,aa1); // send incremented data back

      if len = 2 then                // if  message contains two
                                     //   data bytes
      begin
        portd := data[1];            //   output second
                                     //   byte at portd
      end;
    end;
  end;
end.
```

Example of interfacing CAN transceiver with MCU and bus



CANSPI Library

SPI (Serial Peripheral Interface) module is available with a number of PIC MCU models. Set of library procedures and functions is listed below to provide comfortable work with external CAN modules (such as MCP2515 or MCP2510) via SPI.

CANSPI routines are supported by any PIC MCU model that has SPI interface on portc. Also, CS pin of MCP2510 or MCP2515 must be connected to RC0 pin. Example of HW connection is given at the end of the chapter.

The Controller Area Network module is a serial interface, useful for communicating with other peripherals or microcontrollers. Details about CAN can be found in appropriate literature and on mikroElektronika Web site. MCP2515 or MCP2510 are modules that enable any chip with SPI interface to communicate over CAN bus.

Following routines should be considered a driver for CANSPI (CAN via SPI module) on PIC MCUs.

```
procedure CANSPISetOperationMode(mode : byte, WAIT : byte);

function  CANSPIGetOperationMode : byte;

procedure CANSPIInitialize( SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS : byte);

procedure CANSPISetBaudRate( SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS : byte);

procedure CANSPISetMask( CAN_MASK : byte; val : longint; CAN_CONFIG_FLAGS : byte);

procedure CANSPISetFilter( CAN_FILTER : byte; val : longint;  CAN_CONFIG_FLAGS : byte);

function  RegsToCANSPIID( var ptr : byte;  CAN_CONFIG_FLAGS : byte): longint;

procedure CANSPIIDToRegs( var ptr: byte; val : longint;  CAN_CONFIG_FLAGS : byte);

function  CANSPIwrite(id : longint; var Data : array[8] of byte;
                      DataLen, CAN_TX_MSG_FLAGS: byte) : byte;

function  CANreadSPI( var id : longint; var Data : array[8] of byte; var DataLen: byte;
                       var CAN_RX_MSG_FLAGS : byte) : byte;
```

CANSPISetOperationMode

Prototype: `procedure CANSPISetOperationMode(mode : byte; WAIT : byte);`

Parameters: mode - Operation mode code can take any of predefined constant values (see the constants below)
WAIT - Should have value TRUE(255) or FALSE(0)

Effects: CAN is set to requested mode

Overview: Given mode byte is copied to CANSTAT

Note: If WAIT is true, this is a blocking call. It won't return until requested mode is set.

If WAIT is false, this is a non-blocking call. It does not verify if CAN module is switched to requested mode or not. Caller must use CANSPIGetOperationMode() to verify correct operation mode before performing mode specific operation.

CANSPIGetOperationMode

Prototype: `function CANSPIGetOperationMode : byte;`

Parameters: None

Output: Current operational mode of CAN module is returned

CANSPIInitialize

Prototype:
```
procedure CANSPIInitialize( SJW, BRP, PHSEG1, PHSEG2, PROPSEG,
CAN_CONFIG_FLAGS : byte);
```

Precondition: CAN must be in Configuration mode or else these values will be ignored.

Parameters:
SJW value as defined in 18XXX8 datasheet (must be between 1 thru 4)
BRP value as defined in 18XXX8 datasheet (must be between 1 thru 64)
PHSEG1 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PHSEG2 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PROPSEG value as defined in 18XXX8 datasheet (must be between 1 thru 8)
CAN_CONFIG_FLAGS value is formed from constants (see below)

Effects: CAN bit rate is set. All masks registers are set to '0' to allow all messages.

Filter registers are set according to flag value:

```
If (CAN_CONFIG_FLAGS and CAN_CONFIG_VALID_XTD_MSG) <> 0
    Set all filters to XTD_MSG
Else if (config and CONFIG_VALID_STD_MSG) <> 0
    Set all filters to STD_MSG
Else
    Set half of the filters to STD, and the rest to XTD_MSG.
```

Side Effects: All pending transmissions are aborted.

CANSPISetBaudRate

Prototype: `procedure CANSPISetBaudRate(SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS : byte);`

Precondition: CAN must be in Configuration mode or else these values will be ignored.

Parameters: SJW value as defined in 18XXX8 datasheet (must be between 1 thru 4)
BRP value as defined in 18XXX8 datasheet (must be between 1 thru 64)
PHSEG1 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PHSEG2 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PROPSEG value as defined in 18XXX8 datasheet (must be between 1 thru 8)
CAN_CONFIG_FLAGS - Value formed from constants (see section below)

Effects: CAN bit rate is set as per given values.

Overview: Given values are bit adjusted to fit in 18XXX8. BRGCONx registers and copied.

CANSPISetMask

Prototype: `procedure CANSPISetMask( CAN_MASK : byte; val : longint; CAN_CONFIG_FLAGS : byte);`

Precondition: CAN must be in Configuration mode. If not, all values will be ignored.

Parameters: CAN_MASK - One of predefined constant value
val - Actual mask register value.
CAN_CONFIG_FLAGS - Type of message to filter, either
CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG

Effects: Given value is bit adjusted to appropriate buffer mask registers.

CANSPISetFilter

Prototype:
```
procedure CANSPISetFilter(CAN_FILTER : byte; val : longint;
CAN_CONFIG_FLAGS : byte);
```

Precondition: CAN must be in Configuration mode. If not, all values will be ignored.

Parameters: CAN_FILTER - One of predefined constant values
val - Actual filter register value.
CAN_CONFIG_FLAGS - Type of message to filter, either
CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG

Effects: Given value is bit adjusted to appropriate buffer filter registers.

RegsTOCANSPIID and CANSPIIDToRegs

Prototypes:
```
function RegsToCANSPIID(var ptr : byte; CAN_CONFIG_FLAGS : byte):
longint;

procedure CANSPIIDToRegs(var ptr: byte; val : longint;
CAN_CONFIG_FLAGS : byte);
```

Effects: These two routines are used by other routines (internal purpose only).

CANSPIWrite

Prototype:
```
function CANSPIwrite( id : longint; var Data : array[8] of byte;
DataLen, CAN_TX_MSG_FLAGS : byte) : byte;
```

Precondition: CAN must be in Normal mode.

Parameters: id - CAN message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended).
Data - array of bytes up to 8 bytes in length
DataLen - Data length from 1 thru 8.
CAN_TX_MSG_FLAGS - Value formed from constants (see section below)

Effects: If at least one empty transmit buffer is found, given message is queued for the transmission. If none found, FALSE value is returned.

CANSPIRead

Prototype:
```
function CANSPIRead(var id : longint; var Data : array[8] of byte;
var DataLen : byte; var CAN_RX_MSG_FLAGS : byte) : byte;
```

Precondition: CAN must be in mode in which receiving is possible.

Parameters: id - CAN message identifier
Data - array of bytes up to 8 bytes in length
DataLen - Data length from 1 thru 8.
CAN_TX_MSG_FLAGS - Value formed from constants (see below)

Effects: If at least one full receive buffer is found, it is extracted and returned.
If none found, FALSE value is returned.

CAN Library Constants

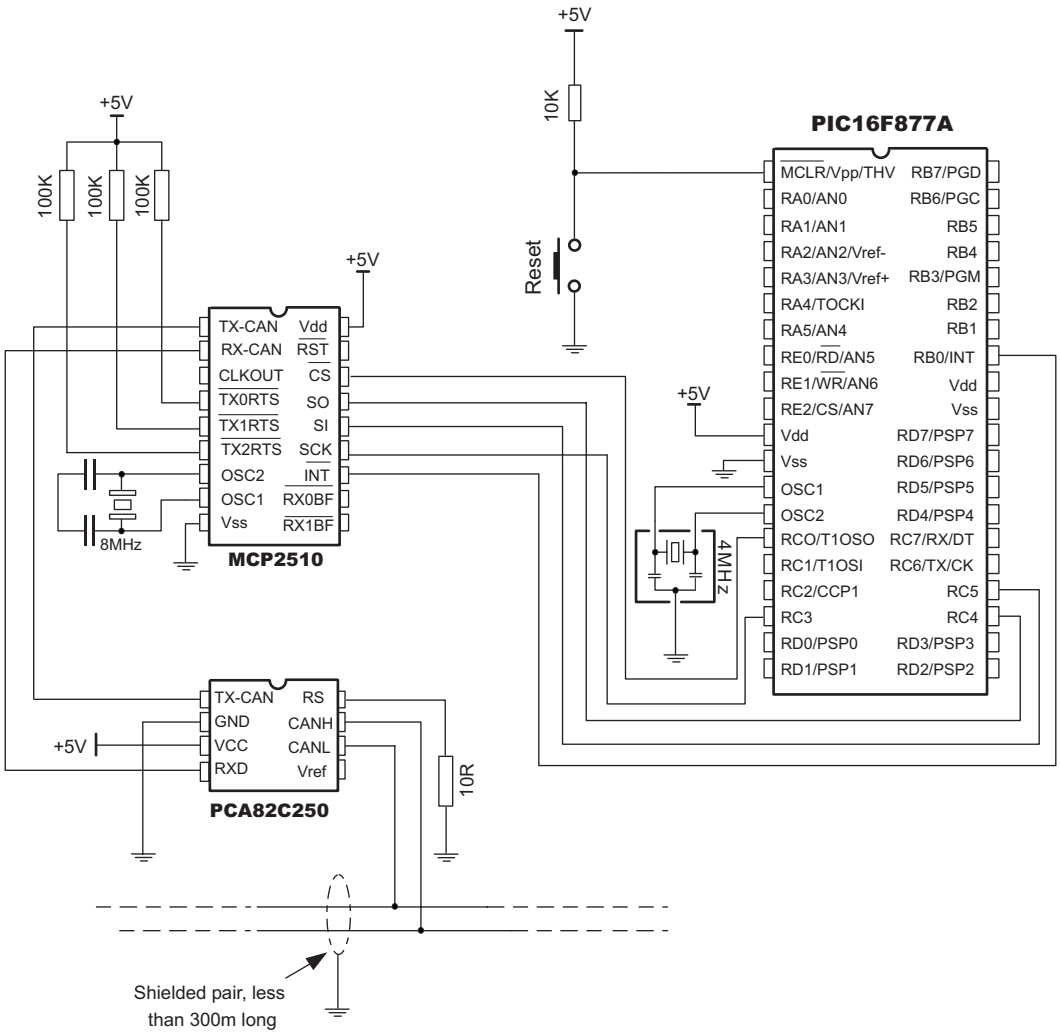
You need to be familiar with constants that are provided for use with CANSPI library routines. See how to form values (from constants) that will be passed to or from routines in the example at the end of the chapter. All of the constants are pre-defined in CAN library.

For the complete list of constants refer to page 119.

Example This code demonstrates use of CANSPI library routines.

```
program CANSPI;

var
    aa, aa1, len, aa2 : byte;
    data : array[8] of byte;
    id : longint;
    zr : byte;

begin
  TRISB := 0;
  SPI_init;          // must be performed before other activities
  ClearBit(TRISC,2); // pin is connected to Reset pin of MCP2510
  ClearBit(portc,2); // keep MCP2510 in reset state
  SetBit(PORTC,0);   // make sure that MCP2510 is not selected
  ClearBit(TRISC,0); // make RC0 output
  PORTD := 0;
  TRISD := 0;        // designate portd as output
  aa := 0;
  aa1 := 0;
  aa2 := 0;
  aa := CAN_CONFIG_SAMPLE_THRICE and
        CAN_CONFIG_PHSEG2_PRG_ON and
        CAN_CONFIG_STD_MSG and
        CAN_CONFIG_DBL_BUFFER_ON and
        CAN_CONFIG_VALID_XTD_MSG;    // prepare flags for
                                     // CANSPIinitialize procedure


  { continues.. }
```

```
  { ..continued }

  SetBit(PORTC,2);        // activate MCP2510 chip
  aa1 := CAN_TX_PRIORITY_BITS and
         CAN_TX_FRAME_BIT and
         CAN_TX_RTR_BIT;
         // prepare flags for CANSPIwrite function


  CANSPIInitialize(1,2,3,3,1,aa);  // initialize MCP2510
  CANSPISetOperationMode(CAN_MODE_CONFIG,true); // set config mode

  ID: = -1;

  CANSPISetMask(CAN_MASK_B1,id,CAN_CONFIG_XTD_MSG);
    // bring all mask1 bits to ones

  CANSPISetMask(CAN_MASK_B2,0,CAN_CONFIG_XTD_MSG);
    // bring all mask2 bits to ones

  CANSPISetFilter(CAN_FILTER_B1_F1,12111,CAN_CONFIG_XTD_MSG);
    // set filter_b1_f1 id to 12111

  CANSPISetOperationMode(CAN_MODE_NORMAL,true);
    // get back to Normal mode

  while true do
  begin
    zr := CANSPIRead(id , Data , len, aa2);
    if (id = 12111) and zr then
    begin
       portd := $AA;
       portB := data[0];
       data[0]:= data[0]+1;
       id:=3;
       delay_ms(10);
       CANSPIWrite(id,data,1,aa1);
      if len = 2 then
      begin
        portd := data[1];
      end;
    end;
  end;
end.
```



Example of interfacing CAN transceiver MCP2551 and MCP2510 with MCU and bus

Compact Flash Library

Compact Flash Library provides routines for accessing data on Compact Flash card (abbrev. CF further in text). CF cards are widely used memory elements, commonly found in digital cameras. Great capacity (8MB ~ 2GB, and more) and excellent access time of typically few microseconds make them very attractive for microcontroller applications.

Following routines can be used for CF with FAT16, and FAT32 file system. Note that routines for file handling can be used only with FAT16 file system.

File accessing routines can write file up to 128KB in size. File names must be exactly 8 characters long and written in uppercase. User must ensure different names for each file, as CF routines will not check for possible match.

In CF card, data is divided into sectors, one sector usually comprising 512 bytes (few older models have sectors of 256B). Read and write operations are not performed directly, but successively through 512B buffer. Before write operation, make sure you don't overwrite boot or FAT sector as it could make your card on PC or digital cam unreadable. Drive mapping tools, such as Winhex, can be of a great assistance.

Following routines implement data and file access to Compact Flash:

```
procedure CF_INIT_PORT(var CtrlPort : byte, var DataPort : byte );
function  CF_DETECT(var CtrlPort : byte) : byte ;
procedure CF_WRITE_INIT(var CtrlPort: byte; var DataPort: byte; Adr: longint; SectCnt: byte);
procedure CF_WRITE_BYTE(var CtrlPort: byte; var DataPort: byte; BData: byte);
procedure CF_WRITE_WORD(var CtrlPort: byte; var DataPort: byte; WData: word);
procedure CF_READ_INIT(var CtrlPort: byte; var DataPort: byte; Adr: longint; SectCnt: byte);
function  CF_READ_BYTE(var CtrlPort: byte; var DataPort: byte) : byte;
function  CF_READ_WORD(var CtrlPort: byte; var DataPort: byte): word;
procedure CF_SET_REG_ADR(var CtrlPort: byte; adr: byte);


procedure CF_File_Write_Init(var CtrlPort : byte; var DataPort : byte);
procedure CF_File_Write_Byte(var CtrlPort : byte; var DataPort : byte; Bdata : byte);
procedure CF_File_Write_Complete(var CtrlPort : byte; var DataPort : byte;
                                 var Filename : array[9] of char);
```

CF_INIT_PORT

Prototype: `procedure CF_INIT_PORT(var CtrlPort : byte, var DataPort : byte );`

Precondition: None.

Parameters: *CtrlPort* is control port, *DataPort* is data port to which CF is attached.

Effects: Initializes ports appropriately.

CF_DETECT

Prototype: `function CF_DETECT(var CtrlPort : byte) : byte;`

Precondition: *CtrlPort* must be initialized (call CF_INIT_PORT first).

Parameters: Check for presence of CF.

Effects: Returns TRUE if CF is present, otherwise returns FALSE.

CF_WRITE_INIT

Prototype:
```
procedure CF_WRITE_INIT(var CtrlPort : byte; var DataPort : byte;
Adr : longint; SectCnt : byte);
```

Precondition: Ports must be initialized.

Parameters: *CtrlPort* - control port , *DataPort* - data port , *Adr* - specifies sector address from where data will be written, *SectCnt* - parameter is total number of sectors prepared for write.

Effects: This procedure initializes CF card for write operation.

CF_WRITE_BYTE

Prototype: `procedure CF_WRITE_BYTE(var CtrlPort : byte; var DataPort : byte; BData : byte);`

Precondition: ports must be initialized, CF must be initialized for write operation (see CF_WRITE_INIT).

Parameters: *CtrlPort* - control port , *DataPort* - data port , *dat* - is data byte written to CF.

Effects: Write 1 byte to CF. This procedure has effect if writing is previously initialized, and all 512 bytes are transferred to a buffer.

CF_WRITE_WORD

Prototype: `procedure CF_WRITE_WORD(var CtrlPort : byte; var DataPort : byte; WData : word);`

Precondition: Ports must be initialized, CF must be initialized for write operation (see CF_WRITE_INIT).

Parameters: *CtrlPort* - control port , *DataPort* - data port , *dat* - is data word written to CF.

Effects: Writes 1 word to CF. This procedure has effect if writing is previously initialized, and all 512 bytes are transferred to a buffer.

CF_READ_INIT

Prototype: `procedure CF_READ_INIT(var CtrlPort : byte; var DataPort : byte; Adr : longint; SectCnt : byte);`

Precondition: Ports must be initialized.

Parameters: *CtrlPort* - control port , *DataPort* - data port , *Adr* - specifies sector address from where data will be read, *SectCnt* - parameter is total number of sectors prepared for read operations.

Effects: This procedure initializes CF card for write operation.

CF_READ_BYTE

Prototype: `function CF_READ_BYTE(var CtrlPort: byte;var DataPort: byte): byte;`

Precondition: Ports must be initialized, CF must be initialized for read operation (see CF_READ_INIT).

Parameters: *CtrlPort* - control port , *DataPort* - data port.

Effects: Read 1 byte from CF.

CF_READ_WORD

Prototype: `function CF_READ_WORD(var CtrlPort: byte;var DataPort: byte): word;`

Precondition: Ports must be initialized, CF must be initialized for read operation (see CF_READ_INIT).

Parameters: *CtrlPort* - control port , *DataPort* - data port.

Effects: Read 1 word from CF.

CF_SET_REG_ADR

Prototype: `procedure CF_SET_REG_ADR(var CtrlPort : byte; adr : byte);`

Effects: This procedure is for internal use only.

CF_FILE_WRITE_INIT

Prototype:
```
procedure CF_File_Write_Init(var CtrlPort : byte;
                             var DataPort : byte);
```

Precondition: Ports must be initialized, CF must be initialized for read operation (see CF_READ_INIT).

Parameters: CtrlPort - control port, DataPort - data port.

Effects: This procedure initializes CF card for file writing operation (FAT16 only).

CF_FILE_WRITE_BYTE

Prototype:
```
procedure CF_File_Write_Byte(var CtrlPort : byte;
                             var DataPort : byte; Bdata : byte);
```

Precondition: Ports must be initialized, CF must be initialized for write operation (see CF_File_Write_Init).

Parameters: CtrlPort - control port, DataPort - data port, Bdata - data byte to be written.

Effects: This procedure adds one byte (<*Bdata*>) to file.

CF_FILE_WRITE_BYTE

Prototype:
```
procedure CF_File_Write_Complete(var CtrlPort : byte;
            var DataPort : byte; var Filename : array[9] of char);
```

Parameters: CtrlPort - control port, DataPort - data port, Filename (**must be in uppercase and must have exactly 8 characters**).

Effects: Upon all data has be written to file, use this procedure to close the file and make it readable by Windows.

Example

The following code demonstrates use of CF library procedures and functions.

```
program CompactFlash;

var i : word;
    temp, k : longint;


begin
  TRISC := 0;                    // designate portc as output
  CF_INIT_PORT(PORTB,PORTD);     // initialize ports

  repeat
     nop;
  until CF_DETECT(PORTB) = true;
    // wait until CF card is inserted


  Delay_ms(500);
  CF_WRITE_INIT(PORTB,PORTD,590,1);
    // Initialize write at sector address 590
    //    of  1 sector (512 bytes)

    // write 512 bytes to sector (590)

  for i := 0 to 511 do
  begin
    CF_WRITE_BYTE(PORTB,PORTD,i+11);
  end;

  PORTC := $FF;
  Delay_ms(1000);
  CF_READ_INIT(PORTB,PORTD,590,1);
    // Initialize write at sector address 590
    //    of  1 sector (512 bytes)

    // read 512 bytes from sector (590)

  for i := 0 to 511 do
  begin
       PORTC := CF_READ_BYTE(PORTB,PORTD);
        // read byte and display on portc
       Delay_ms(1000);
  end;
end.
```

Example

```
program CompactFlash_File;

var    i1 : word;
       index: byte;
       Fname: string[9];

procedure Init;
begin
 TRISC := 0;                        // designate portc as output
 CF_Init_Port(PORTB,PORTD);         // initialize ports
 repeat
    nop;
  until CF_Detect(PORTB) = TRUE; // wait until CF card is inserted
 Delay_ms(50);                      // wait until the card stabilizes
end;

begin

  index := 0;                       // index of file to be written
  while index < 5 do
    begin
     portc := 0;
     Init;
     portc:= index;
     CF_File_Write_Init(PORTB,PORTD);  // initialization for
                                       // writing to new file
     i1 := 0;
     while i1 < 50000 do
     begin
       CF_File_Write_Byte(PORTB,PORTD,48+index);  // writes 50000
                                                  // bytes to file
       inc(i1);
     end;
     Fname := 'RILEPROX';    // must be 8 character long
                             // all in upper case

     fname[8] := 48+index;   // ensure that files have
                             // different names

     CF_File_Write_Complete(PORTB,PORTD, Fname); // closes the file
     Inc(index);
   end;
   PORTC:=$FF;

end.
```

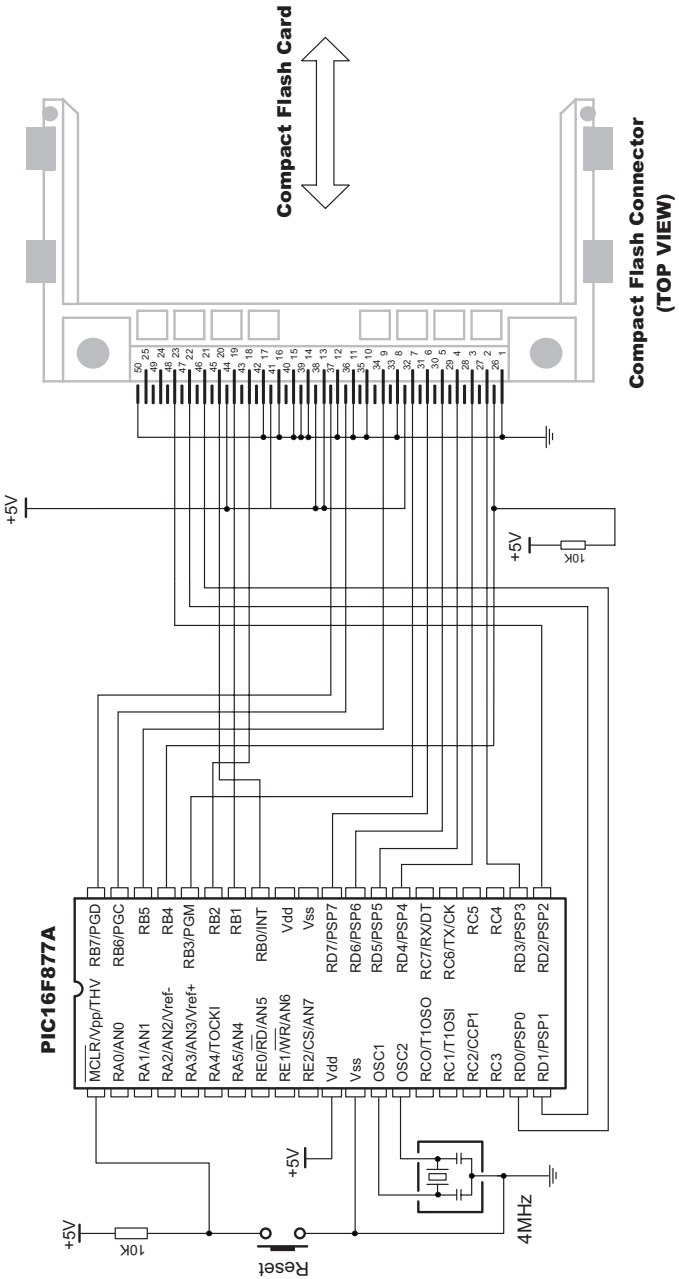


Figure: Pin diagram of Compact Flash memory card

EEPROM Library

EEPROM data memory is available with a number of PIC MCU models. Set of library procedures and functions is listed below to provide you comfortable work with EEPROM.

Routines

Basically, there are two operations that can be performed on EEPROM data memory.

```
function  EEprom_Read(Address: byte): byte;
procedure EEprom_Write(Address, Data: byte);
```

Library function EEprom_Read reads data from specified Address, while the library procedure EEprom_Write writes Data to specified Address.

Note

Parameter *Address* is of byte type, which means it can address only 256 locations. For PIC18 MCU models with more EEPROM data locations, it is programmer's responsibility to set SFR EEADRH register appropriately.

Example

```
program EEPROMtest;
var i,j : byte;

begin
  TRISB := 0;
  for i:=0 to 20 do
   EEprom_write(i,i+6);

  for i := 0 to 20 do
   begin
     PORTB := EEprom_read(i);
     for j := 0 to 200 do
       delay_us(500);
   end;
end.
```

I2C Library

I2C (Inter Integrated Circuit) full master MSSP (Master Synchronous Serial Port) module is available with a number of PIC MCU models. Set of library procedures and functions is listed below to support the master I2C mode.

Important

Note that these functions support module on PORTC, and won't work with modules on other ports. Examples for PIC MCUs with module on other ports can be found in your mikroPascal installation folder, subfolder 'examples'.

Routines

I2C interface is serial interface used for communicating with peripheral or other microcontroller devices. All functions and procedures bellow are intended for PIC MCUs with MSSP module. By using these, you can configure and use PIC MCU as master in I2C communication.

```
procedure I2C_Init(const clock: longint);
function  I2C_Is_Idle: byte;
function  I2C_Start: byte;
procedure I2C_Repeated_Start;
function  I2C_Wr(Data: byte): byte;
function  I2C_Rd(Ack: byte): byte;
procedure I2C_Stop;
```

```
procedure I2C_Init(const clock: longint);
```

Parameter *clock* is a desired I2C clock (refer to device data sheet for correct values in respect with Fosc).

Example:

```
I2C_init(100000);
```

After configuring the I2C master mode, you have the following functions and procedures at your disposal:

```
function I2C_Start: byte;
```

Determines if I2C bus is free and issues START condition; if there is no error, function returns 0.

```
procedure I2C_Repeated_Start;
```

Performs repeated start condition.

```
function I2C_Wr(Data: byte): byte;
```

After you have issued a start or repeated start you can send data byte via I2C bus; this function also returns 0 if there is no errors.

```
function I2C_Rd(Ack: byte): byte;
```

Receives 1 byte from the slave; and sends not acknowledge signal if parameter Ack is 0 in all other cases it sends acknowledge.

```
procedure I2C_Stop;
```

Issues STOP condition.

Example

The following code demonstrates use of I2C Library procedures and functions. PIC MCU is connected (SCL,SDA pins) to 24c02 EEPROM. Program sends data to EEPROM (data is written at address 2). Then, we read data via I2C from EEPROM and send its value to PORTD, to check if the cycle was successful. See the following figure on how to interface 24c02 to PIC.

```
{ Example of communication with 24c02 EEPROM }

program pascalI2c;

var  EE_adr, EE_data : byte;
     jj : word;

begin
    I2C_init(100000);      // initialize full master mode
    TRISD := 0;            // designate portd as output
    PORTD := $ff;          // initialize portd
    I2C_Start;             // issue I2C start signal
    I2C_Wr($a2);           // send byte via I2C(command to 24cO2)
    EE_adr := 2;
    I2C_Wr(EE_adr);        // send byte(address for EEPROM)
    EE_data := $aa;
    I2C_Wr(EE_data);       // send data to be written
    I2C_Stop;              // issue I2C stop signal

    for jj:=0 to 65500 do
    nop;                   // pause while EEPROM writes data

    I2C_Start;             // issue I2C start signal
    I2C_Wr($a2);           // send byte via I2C
    EE_adr := 2;
    I2C_Wr(EE_adr);        // send byte(address for EEPROM)
    I2C_Repeated_Start;    // issue I2Csignal repeated start
    I2C_Wr($a3);           // send byte(request data from EEPROM)
    k := I2C_rd(1);        // read the data
    I2C_Stop;              // issue I2C_stop signal
    PORTD := k;            // show data on PORTD
while true do              // endless loop
  nop;
end.
```

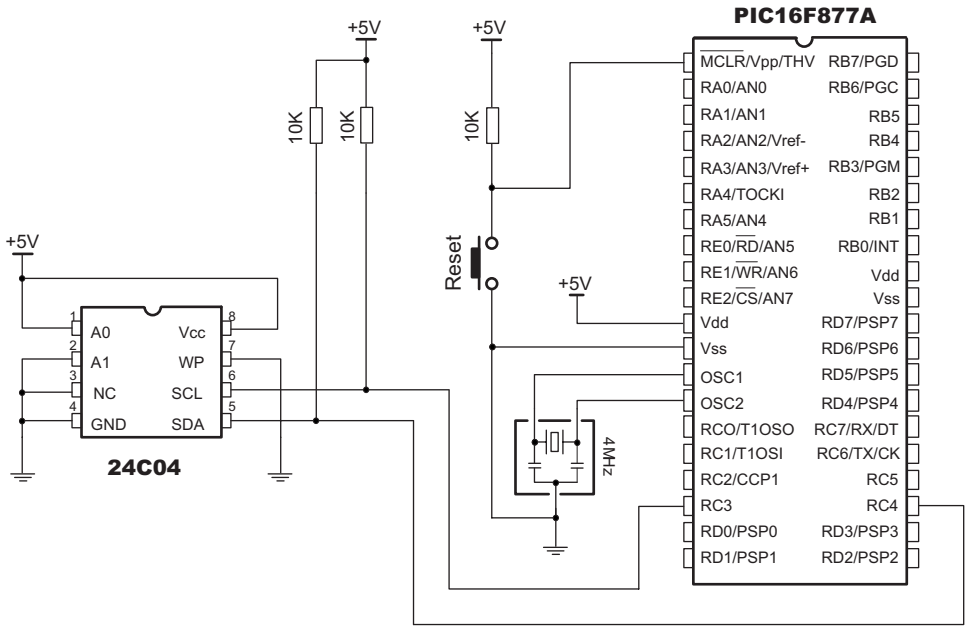


Figure: I2C interfacing EEPROM 24C04 to PIC MCU

LCD Library

mikroPascal provides a set of library procedures and functions for communicating with commonly used LCD (4-bit interface). Figure showing HW connection of PIC and LCD is given at the end of the chapter (if you need different pin settings, refer to LCD_Config routine).

Note

Be sure to designate port with LCD as output, before using any of the following library routines.

Routines

```
procedure LCD_Config(var Port: byte;
                     const RS, EN, WR, D7, D6, D5, D4 : byte);
```

Initializes LCD at <*Port*> with pin settings you specify: parameters <*RS*>, <*EN*>, <*WR*>, <*D7*> .. <*D4*> need to be a combination of values 0..7 (e.g. 3,6,0,7,2,1,4).

```
procedure LCD_Init(var Port : byte);
```

Initializes LCD at <*Port*> with default pin settings (check the figures at the end of the chapter).

```
procedure LCD_Out(Row, Column: byte; var Text: array[255] of char);
```

Prints <*Text*> (string variable) at specified row and column on LCD. Both string variables and string constants can be passed.

```
procedure LCD_Out_CP(var Text: array[255] of char);
```

Prints <*Text*> (string variable) at current cursor position. Both string variables and string constants can be passed.

```
procedure LCD_Chr(Row, Column, Character: byte);
```

Prints <*Character*> at specified row and column on LCD.

```
procedure LCD_Chr_CP(Character: byte);
```

Prints <*Character*> at current cursor position.

```
procedure LCD_Cmd(Command: byte);
```

Sends command <*Command*> to LCD. Refer to the following list of available commands.

LCD Commands

Command	Purpose
LCD_First_Row	Moves cursor to 1st row
LCD_Second_Row	Moves cursor to 2nd row
LCD_Third_Row	Moves cursor to 3rd row
LCD_Fourth_Row	Moves cursor to 4th row
LCD_Clear	Clears display
LCD_Return_Home	Returns cursor to home position, returns a shifted display to original position. Display data RAM is unaffected
LCD_Cursor_Off	Turns off cursor
LCD_Underline_On	Underline cursor on
LCD_Blink_Cursor_On	Blink cursor on
LCD_Move_Cursor_Left	Move cursor left without changing display data RAM
LCD_Move_Cursor_Right	Move cursor right without changing display data RAM
LCD_Turn_On	Turn LCD display on
LCD_Turn_Off	Turn LCD display off
LCD_Shift_Left	Shift display left without changing display data RAM
LCD_Shift_Right	Shift display right without changing display data RAM

This is a complete list of available LC commands for use with LCD_Cmd and LCD8_Cmd routines.

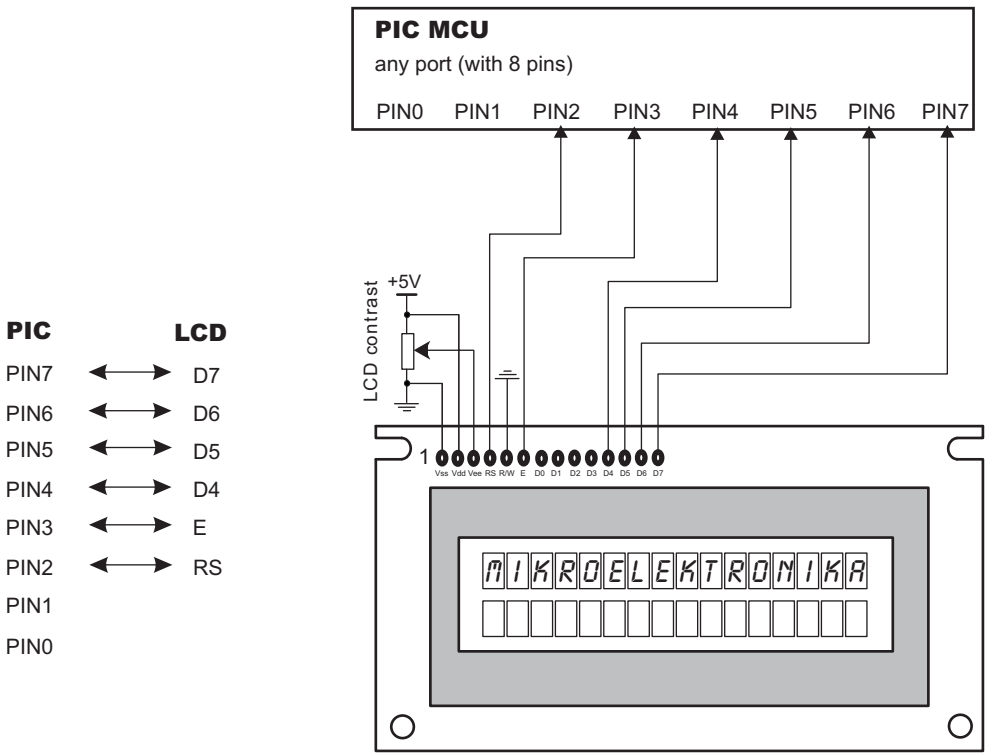
Example Here are several examples of LCD routine calls:

```
LCD_Config(PORTD,0,1,2,6,5,4,3);
 // Initializes LCD on PORTD with custom pin settings
 // (4-bit interface).


LCD_Init(PORTB);
 // Initializes LCD on PORTB with default pin settings
 // (4-bit interface).


LCD_Out(1,1,txt);
 // Prints string variable <txt> on LCD (1st row, 1st column).


LCD_Out_CP(txt);
 // Prints string variable <txt> at current cursor position.


LCD_Char(1,1,'e');
 // Prints character 'e' on LCD (1st row, 1st column).


LCD_Char_CP('f');
 // Prints character 'f' at current cursor position.


LCD_Cmd(LCD_Clear);
 // Sends command LCD_Clear to LCD (clears LCD display).
```

Example

Use LCD_Init for default pin settings (see the figure below).

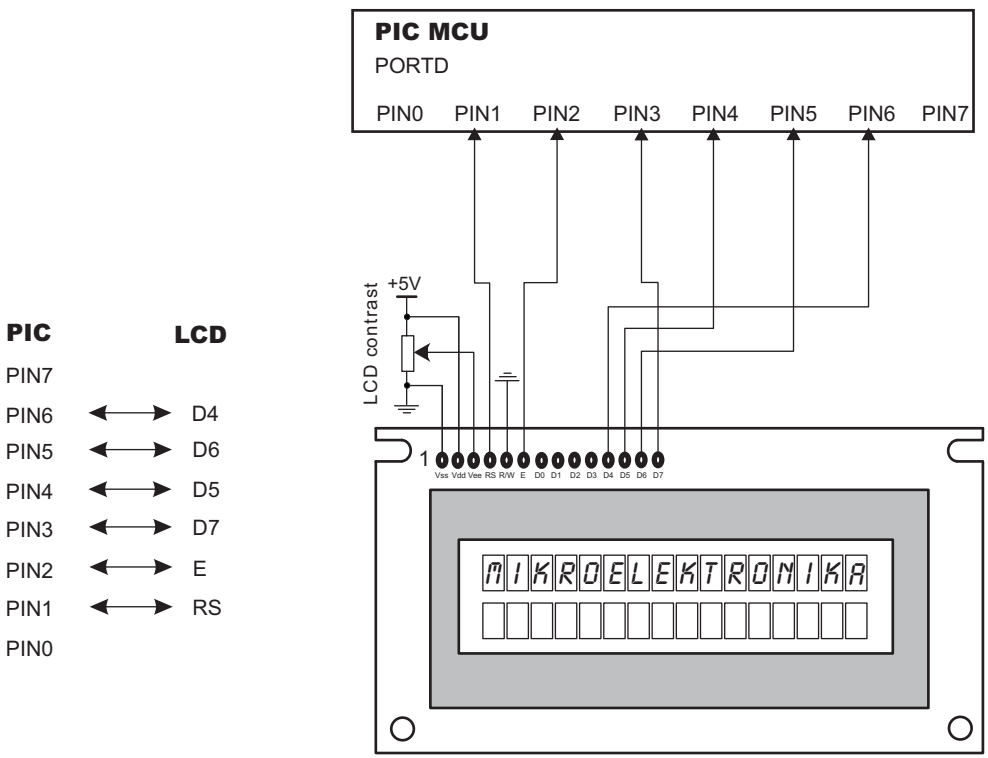
```
program LCD_default_test;


begin
  TRISB := 0;                              // Portb is output
  LCD_Init(PORTB);                         // Initialize LCD on portb
  LCD_Cmd(LCD_CURSOR_OFF);                 // Turn off cursor
  LCD_Out(1, 1, 'mikroelektronika');       // Print text at LCD
end.
```



LCD HW connection by default initialization (using LCD_Init). If you need different pin settings, refer to LCD_Config routine.

Alternatively, you can use LCD_Config for custom pin settings. For example:

```
program LCD_custom_test;


begin
  TRISD := 0;                              // Portd is output

  // Initialize LCD on portd with custom pin settings
  LCD_Config(PORTD,1,2,0,3,5,4,6);
  LCD_Cmd(LCD_CURSOR_OFF);                 // Turn off cursor
  LCD_Out(1, 1, 'mikroelektronika');       // Print text at LCD
end.
```



LCD HW custom connection (using LCD_Config, see the program above).

LCD8 Library (8-bit interface LCD)

mikroPascal provides a set of library procedures and functions for communicating with commonly used 8-bit interface LCD (with Hitachi HD44780 controller). Figure showing HW connection of PIC and LCD is given at the bottom of the page (if you need different pin settings, refer to LCD8_Config routine).

Note

Be sure to designate Control and Data ports with LCD as output, before using any of the following library routines.

Routines

```
procedure LCD8_Config(var portCtrl : byte; var portData : byte;
         const RS, EN, WR, D7, D6, D5, D4, D3, D2, D1, D0 : byte);
```

Initializes LCD at <*portCtrl*> and <*portData*> with pin settings you specify:
parameters <*RS*>, <*EN*>, <*WR*> need to be in range 0..7;
parameters <*D7*> .. <*D0*> need to be a combination of values 0..7
(e.g. 3,6,5,0,7,2,1,4).

```
procedure LCD8_Init(var portCtrl : byte; var portData : byte);
```

Initializes LCD at <*portCtrl*> and <*portData*> with default pin settings (check the figures at the end of the chapter).

```
procedure LCD8_Out(Row : byte, Column : byte;
                   var Text : array[255] of char);
```

Prints <*Text*> (string variable) at specified row and column on LCD. Both string variables and string constants can be passed.

```
procedure LCD8_Out_CP(var Text : array[255] of char);
```

Prints <*Text*> (string variable) at current cursor position. Both string variables and string constants can be passed.

```
procedure LCD8_Chr(Row : byte, Column, Character : byte);
```

Prints <*Character*> at specified row and column on LCD.

```
procedure LCD8_Chr_CP(Character : byte);
```

Prints <*Character*> at current cursor position.

```
procedure LCD8_Cmd(Command : byte);
```

Sends command <*Command*> to LCD. Refer to page 151 for the complete list of available LCD commands.

Example

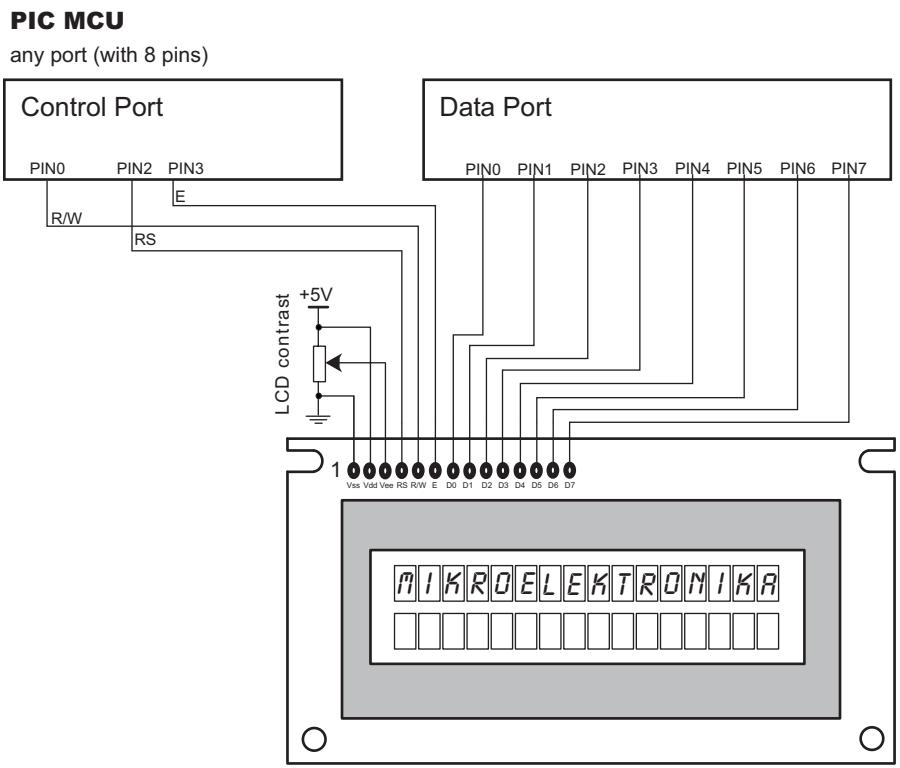
Here are several examples of LCD8 routine calls:

```
LCD8_Config(PORTC,PORTD,0,1,2,6,5,4,3,7,1,2,0);
 // Initializes LCD on PORTC and PORTD with custom pin settings
 // (8-bit interface).


LCD8_Init(PORTB,PORTC);
 // Initializes LCD on PORTB and PORTC with default pin settings
 // (8-bit interface).


LCD8_Out(1,1,txt);
 // Prints string variable <txt> on LCD (1st row, 1st column).


LCD8_Out_CP(txt);
 // Prints string variable <txt> at current cursor position.


LCD8_Char(1,1,'e');
 // Prints character 'e' on LCD (1st row, 1st column).


LCD8_Char_CP('f');
 // Prints character 'f' at current cursor position.


LCD8_Cmd(LCD_Clear);
 // Sends command LCD_Clear to LCD (clears LCD display).
```

Example

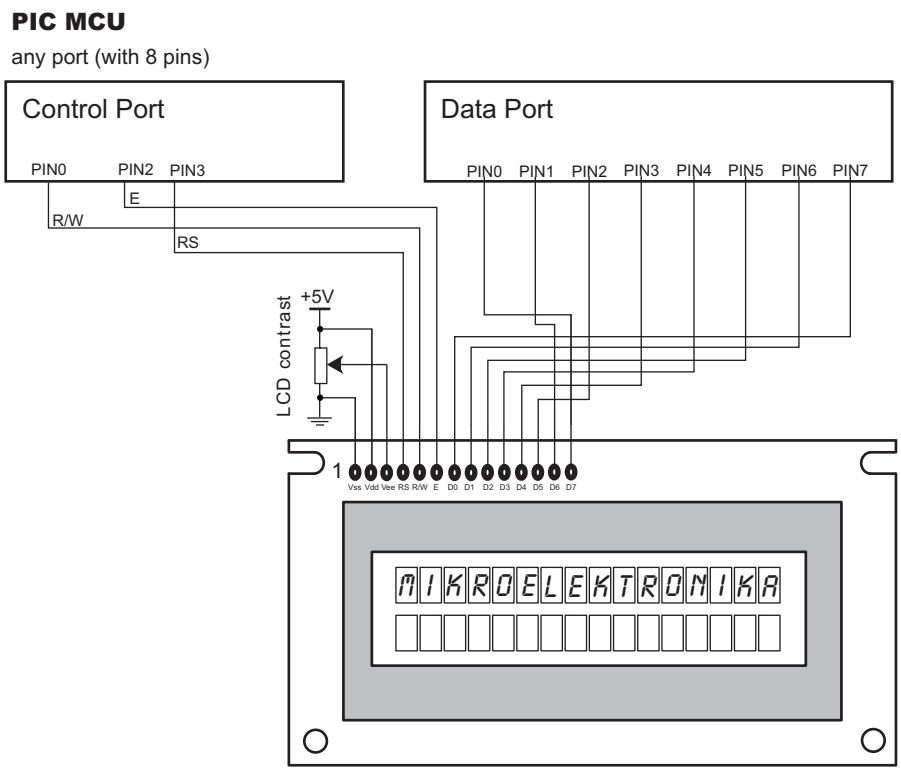
Use LCD8_Init for default pin settings (see the figure below).

```
program LCD8_default_test;

begin
  TRISB := 0;                                    // PORTB is output
  TRISC := 0;                                    // PORTC is output
  LCD8_Init(PORTB, PORTC);    // Initialize LCD at PORTB and PORTC
  LCD8_Cmd(LCD_CURSOR_OFF);                      // Turn off cursor
  LCD8_Out(1, 1, 'mikroElektronika');            // Print Text at LCD
end.
```



Alternatively, you can use LCD8_Config for custom pin settings. For example:

```
program LCD8_custom_test;


begin
  TRISB := 0;                                    // Portb is output
  TRISD := 0;                                    // Portd is output

  // Initialize LCD at portb and portd with custom pin settings
  LCD8_Config(PORTB,PORTD,3,2,0,0,1,2,3,4,5,6,7)

  LCD8_Cmd(LCD_CURSOR_OFF);                      // Turn off cursor
  LCD8_Out(1, 1, 'mikroElektronika');            // Print Text at LCD
end.
```



Graphic LCD Library

mikroPascal provides a set of library procedures and functions for drawing and writing on Graphical LCD. Also it is possible to convert bitmap (use menu option Tools > BMP2LCD) to constant array and display it on GLCD. These routines work with commonly used GLCD 128x64, and work only with the PIC18 family.

Note

Be sure to designate ports with GLCD as output, before using any of the following library procedures or functions.

Routines

```
procedure GLCD_LCD_Init(var Ctrl_Port, Data_Port: byte);
```

Initializes LCD at <*Ctrl_Port*> and <*Data_Port*>.

```
procedure GLCD_LCD_Config(var Ctrl_Port, Data_Port: byte;
                          Reset, Enable, RS, RW, CS1, CS2: byte);
```

Initializes LCD at <*Ctrl_Port*> and <*Data_Port*> with custom pin settings. For example: `GLCD_Config(portb, portc, 1,7,4,6,0,2);`

```
procedure GLCD_Put_Ins(ins : byte);
```

Sends instruction <*ins*> to GLCD. Available instructions include:

```
X_ADRESS     : byte = $B8;       // Adress base for Page 0
Y_ADRESS     : byte = $40;       // Adress base for Y0
START_LINE   : byte = $C0;       // Adress base for line 0
DISPLAY_ON   : byte = $3F;       // Turn display on
DISPLAY_OFF  : byte = $3E;       // Turn display off
```

```
procedure GLCD_Put_Data(data : byte);
```

Sends data byte to GLCD.

```
procedure GLCD_Put_Data2(data, side : byte);
```

Sends data byte to GLCD at specified <*side*>.

```
procedure GLCD_Select_Side(LCDSide : byte);
```

Selects the side of the GLCD:

```
// const RIGHT : byte =  0;
// const LEFT  : byte =  1;
```

```
function GLCD_Data_Read : byte;
```

Reads data from GLCD.

```
procedure GLCD_Set_Dot(x, y : byte);
```

Draws a dot on the GLCD.

```
procedure GLCD_Clear_Dot(x, y : byte);
```

Clears a dot on the GLCD.

```
procedure GLCD_Circle(CenterX, CenterY, Radius : integer);
```

Draws a circle on the GLCD, centered at <*CenterX, CenterY*> with <*Radius*>.

```
procedure GLCD_Line(x1, y1, x2, y2 : integer);
```

Draws a line from (x1,y1) to (x2,y2).

```
procedure GLCD_Invert(Xaxis, Yaxis : byte);
```

Procedure inverts display (changes dot state on/off) in the specified area, X pixels wide starting from 0 position, 8 pixels high. Parameter X spans 0..127, parameter Y spans 0..7 (8 text lines).

```
procedure GLCD_Goto_XY(x, y : byte);
```

Sets cursor to dot (x,y). Procedure is used in combination with GLCD_Put_Data, GLCD_Put_Data2, and GLCD_Put_Char.

```
procedure GLCD_Put_Char(Character : byte);
```

Prints <*Character*> at cursor position.

```
procedure GLCD_Clear_Screen;
```

Clears the GLCD screen.

```
procedure GLCD_Put_Text(x_pos, y_pos: word;
                    var text: array[25] of char; invert: byte);
```

Prints <*text*> at specified position. y_pos spans 0..7.

```
procedure GLCD_Rectangle(X1, Y1, X2, Y2 : byte);
```

Draws a rectangle on the GLCD. (x1,y1) sets the upper left corner, (x2,y2) sets the lower right corner.

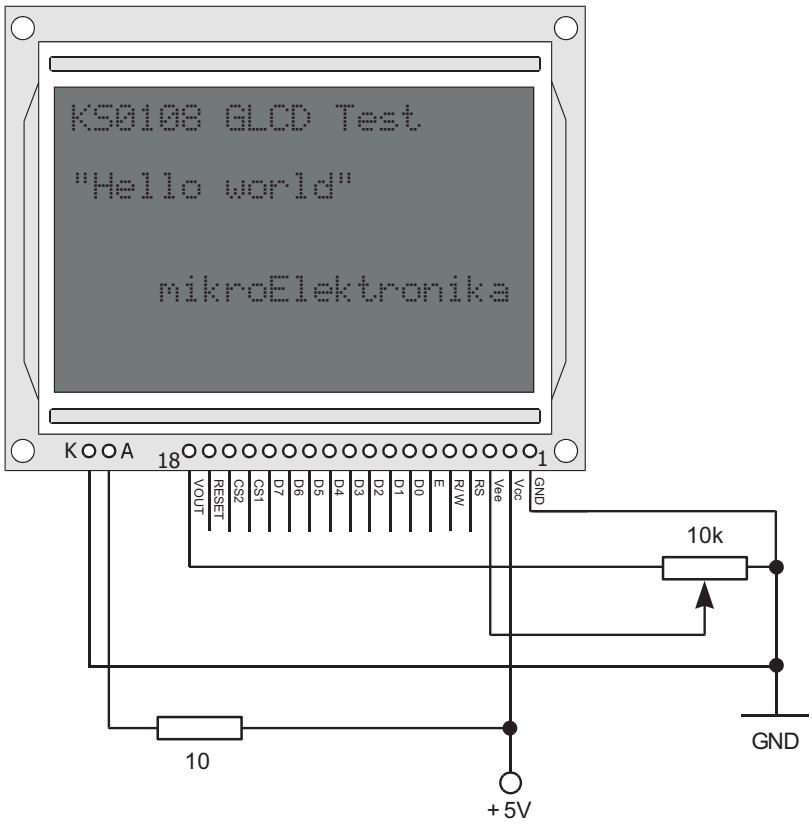
```
procedure GLCD_Set_Font(font_index: byte);
```

Sets font for GLCD. Parameter <*font_index*> spans from 1 to 4, and determines which font will be used: 1: 5x8 dots, 2: 5x7 dots, 3: 3x6 dots, 4: 8x8 dots.

Example

```
program GLCD_test;

uses GLCD_128x64;

var
     text: array[25] of char;
     j, k : byte;

begin
  PORTC := 0; PORTB := 0; PORTD := 0;
  TRISC := 0; TRISB := 0; TRISD := 0;

  GLCD_LCD_Init(PORTC, PORTD);
  GLCD_Set_Font(FONT_NORMAL1);

while true do
    begin
      GLCD_Clear_Screen;          // clear display
      text:='Circles';
      GLCD_Put_Text(0, 7, text, NONINVERTED_TEXT);
      GLCD_Circle(63,31,20);      // draw rectangle
      Delay_Ms(4000);

      GLCD_Invert(127,0);         // inverts the first GLCD line
      Delay_Ms(4000);


      GLCD_Clear_Screen;          // clear display
      text:='Rectangles';
      GLCD_Put_Text(0, 7, text, NONINVERTED_TEXT);
      GLCD_Rectangle(10, 0, 30, 35);   // draw rectangle
      Delay_Ms(4000);

      GLCD_Clear_Screen;          // clear display
      text:='Lines';
      GLCD_Put_Text(0, 7, text, NONINVERTED_TEXT);
      GLCD_Line(0, 0, 127, 50);  // draw line
      GLCD_Line(0,63, 50, 0);    // draw line
      Delay_Ms(5000);

  { continued.. }
```

```
{ ..continued }

      GLCD_Clear_Screen;
      text:='Font DEMO';
      GLCD_Set_Font(FONT_TINY);
      GLCD_Put_Text(0, 4, text, NONINVERTED_TEXT);
      GLCD_Put_Text(0, 5, text, INVERTED_TEXT);
      GLCD_Set_Font(FONT_BIG);
      GLCD_Put_Text(0, 6, text, NONINVERTED_TEXT);
      GLCD_Put_Text(0, 7, text, INVERTED_TEXT);
      Delay_Ms(5000);
    end;
end.
```



PWM Library

CCP (Capture/ Compare/ PWM) module is available with a number of PIC MCU models. Set of library procedures and functions is listed below to provide comfortable work with PWM (Pulse Width Modulation).

Note

Note that these routines support module on PORTC pin RC2, and won't work with modules on other ports. Also, mikroPascal doesn't support enhanced PWM modules. Examples for PIC MCUs with module on other ports can be found in your mikroPascal installation folder, subfolder 'examples'.

Routines

```
procedure PWM_Init(const PWM_Freq);
procedure PWM_Change_Duty(New_Duty: byte);
procedure PWM_Start;
procedure PWM_Stop;
```

```
procedure PWM_Init(const PWM_Freq);
```

Initializes the PWM module. It starts with (duty ratio) 0%.
Parameter *PWM_Freq* is a desired PWM frequency (refer to device data sheet for correct values in respect with Fosc).

Example: `PWM_Init(5000);`

```
procedure PWM_Change_Duty(New_Duty: byte);
```

Parameter *New_Duty* (duty ratio) takes values from 0 to 255, where 0 is 0% duty ratio, 127 is 50% duty ratio, and 255 is 100% duty ratio. Other values for specific duty ratio can be calculated as (Percent*255)/100.

```
procedure PWM_start;
```

Starts PWM.

```
procedure PWM_stop;
```

Stops PWM.

Example

This code demonstrates use of PWM library procedures and functions. If pin RC2 is connected to LED diode, light emitted will depend of PWM duty ratio and this change can be noticed.

```
program PWMtest;

var j : byte;

begin
      j := 0;
      PORTC := $FF;
      PWM_init(5000);       // initializes PWM module, freq = 5kHz
      PWM_start;            // starts PWM
      while true do
      begin
       for i := 0 to 20 do
           delay_ms(100);
       j := j + 1;
       PWM_change_duty(j);  // changes duty ratio
      end;
end.
```

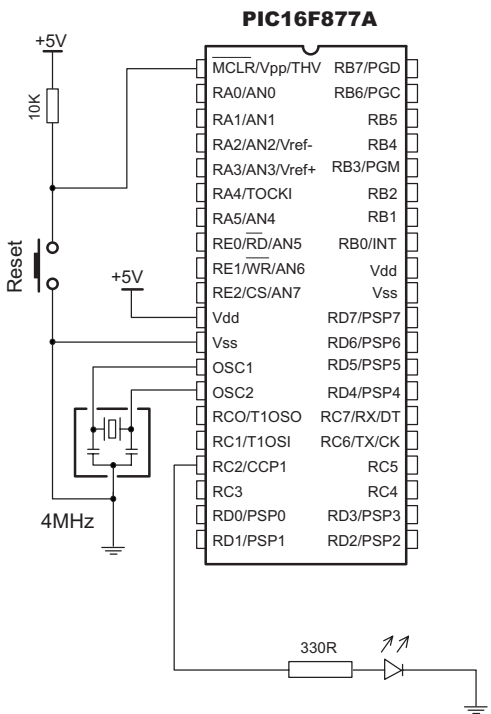


Figure: PWM demonstration

RS485 Library

RS485 is a multipoint communication which allows multiple devices to be connected to a single signal cable. mikroPascal provides a set of library routines to provide you comfortable work with RS485 system using Master/Slave architecture.

Master and Slave devices interchange packets of information, each of these packets containing synchronization bytes, CRC byte, address byte, and the data. In Master/Slave architecture, Slave can never initiate communication. Each Slave has its unique address and receives only the packets containing that particular address. It is programmer's responsibility to ensure that only one device transmits data via 485 bus at a time.

Address 50 is a common address for all Slave devices: packets containing address 50 will be received by all Slaves. The only exceptions are Slaves with addresses 150 and 169, which require their particular address to be specified in the packet.

Note

RS485 routines require USART module on port C. Pins of USART need to be attached to RS485 interface transceiver, such as LTC485 or similar. Pins of transceiver (Receiver Output Enable and Driver Outputs Enable) should be connected to port C, pin 2 (see the figure at end of the chapter).

Routines

Following routines implement flexible protocol for RS485 system with Master/Slave architecture:

```
procedure RS485master_init;
procedure RS485slave_init(address: byte);
procedure RS485slave_read(var data: array[5] of byte);
procedure RS485slave_write(var data: array[2] of byte; datalen: byte);
procedure RS485master_write(var data: array[2] of byte;
                            datalen, address : byte);
procedure RS485master_read(var data: array[5] of byte);
```

RS485master_init

Prototype: `procedure RS485master_init;`

Precondition: USART needs to be initialized (USART_init)

Parameters: None

Effects: Initializes MCU as Master in RS485 communication

RS485slave_init

Prototype: `procedure RS485slave_init(address : byte);`

Precondition: USART needs to be initialized (USART_init)

Parameters: Slave address can take any value between 0 and 255, except 50, which is common address for all slaves)

Effects: Initializes MCU as Slave in RS485 communication

RS485master_read

Prototype: `procedure RS485master_read(var data : array[5] of byte);`

Precondition: MCU must be initialized as Master to assign an address to MCU.

Parameters: `var data : array[5] of byte`

Effects: Master receives any message sent by Slaves. As messages are multi-byte, this procedure must be called for each byte received. Upon receiving a message, buffer is filled with the following values:

data[0..2] is data; data[3] is the number of received bytes (1..3); data[4] is set to 255 (TRUE) when message is received; data[5] is set to 255 (TRUE) if an error has occurred; data[6] is the address of the Slave which sent the message

Procedure automatically sets data[4] and data[5] upon every received message.These flags need to be cleared repeatedly from the program.

RS485master_write

Prototype: `procedure RS485master_write(var data : array[2] of byte; datalen, address : byte);`

Precondition: MCU must be initialized as Master in 485 communication. It is programmer's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.

Parameters: `var data : array[2] of byte, datalen : byte`

Effects: Sends number of bytes (1 < datalen <= 3) from buffer via 485.

RS485slave_read

Prototype: `procedure RS485slave_read(var data : array[5] of byte);`

Precondition: MCU must be initialized as Slave in 485 communication.

Parameters: `var data : array[5] of byte`

Effects: Only messages that appropriately address Slaves will be received. As messages are multi-byte, this procedure must be called for each byte received (see the example at the end of the chapter). Upon receiving a message, buffer is filled with the following values:

data[0..2] is data; data[3] is number of bytes received (1..3) ; data[4] is set to 255(TRUE) when message is received; data[5] is set to 255(TRUE) if an error has occurred; rest of the buffer is undefined

Procedure automatically sets data[4] and data[5] upon every received message. These flags need to be cleared repeatedly from the program.

RS485slave_write

Prototype: `procedure RS485slave_write(var data : array[2] of byte; datalen : byte);`

Precondition: MCU must be initialized as Slave in 485 communication.

Parameters: `var data : array[2] of byte; datalen : byte`

Effects: Sends number of bytes (1 < datalen <= 3) from buffer via 485

Example

```
program pr485;

var dat : array[8] of byte;  // buffer for receiving/sending msgs
    i,j : byte;

procedure interrupt;
begin
 if TestBit(RCSTA,OERR) = 1 then portd := $81;
 slave485_receive(dat);           // every byte is received by
end;                              //    RS485slave_read(dat);
                                  // upon receiving a message
                                  //    without errors
                                  //     data[4] is set to 255
begin
  trisb := 0;
  trisd := 0;
  USART_init(9600);               // initialize usart module
  slave485_init(160);             // initialize MCU as Slave with
                                  //    address 160
  SetBit(PIE1,RCIE);              // enable interrupt
  SetBit(INTCON,PEIE);            //    on byte received
  ClearBit(PIE2,TXIE);            //    via USART (RS485)
  SetBit(INTCON,GIE);
  portb := 0;
  portd := 0;                     // ensure that message received
                                  //    flag is 0
  dat[4] := 0;                    // ensure that error flag is 0
  dat[5] := 0;

  while true do
  begin
    if dat[5] then portd :=$aa;   // if there is an error,
                                  //    set portd to $aa

    if dat[4] then                // if message is received:
    begin
    dat[4] := 0;                  //  clear message received flag
      j := dat[3];                //  number of data bytes received
      for i := 1 to j do
      begin
        portb := dat[i-1];        //  output received data bytes
      end;
      dat[0] := dat[0] + 1;       //  increment received dat[0]
      slave485_send(dat,1);       //  send it back to Master
    end;
  end;
end.
```

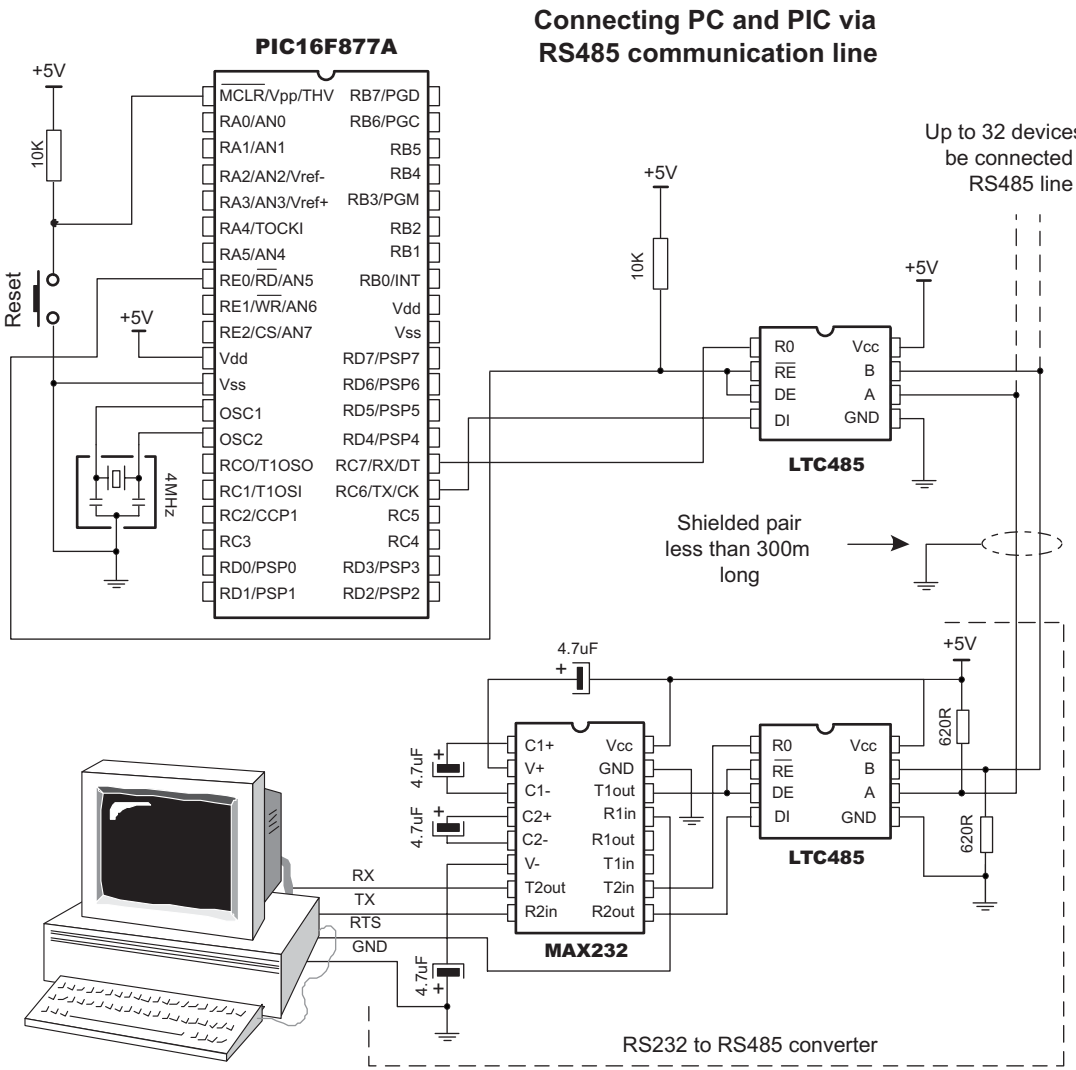


Figure: Example of interfacing PC to PIC MCU via RS485 bus

SPI Library

SPI (Serial Peripheral Interface) module is available with a number of PIC MCU models. Set of library procedures and functions is listed below to provide initialization of slave mode and comfortable work with the master mode.

You can easily communicate with other devices via SPI - A/D converters, D/A converters, MAX7219, LTC1290 etc. You need PIC MCU with hardware integrated SPI (for example, PIC16F877). Then, simply use the following functions and procedures.

Note

Note that these functions support module on PORTB or PORTC, and won't work with modules on other ports. Examples for PIC MCUs with module on other ports can be found in your mikroPascal installation folder, subfolder 'examples'.

Routines

```
procedure SPI_init;
procedure SPI_write(Data: byte);
function  SPI_read(Buffer: byte): byte;
procedure SPI_Init_advanced(Master,Data_Sample,Clock_Idle,Low_To_High:byte);
```

Initialization

You can use procedure `SPI_init` without parameters and get the default result:

Master mode, clock Fosc/4, clock idle state low, data transmitted on low to high edge, input data sampled at the middle of interval;

For advanced settings, configure and initialize SPI using the procedure:

```
procedure SPI_Init_advanced(Master, Data_Sample,
                            Clock_Idle, Low_To_High: byte);
```

Example:

```
SPI_init(Master_OSC_div4, Data_SAMPLE_MIDDLE,LK_Idle_LOW,LOW_2_HIGH)
```

This will set SPI to master mode, clock = Fosc/4, data sampled at the middle of interval, clock idle state low and data transmitted at low to high edge.

Parameters

Parameter *mast_slav* determines the work mode for SPI; can have the following values:

Value	Meaning
Master_OSC_div4	Master clock=Fosc/4
Master_OSC_div16	Master clock=Fosc/16
Master_OSC_div64	Master clock=Fosc/64
Master_TMR2	Master clock source TMR2
Slave_SS_ENABLE	Master slave select enabled
Slave_SS_DIS	Master slave select disabled

Parameter *Data_sample* determines when data is sampled. It can have the following values:

Value	Meaning
Data_SAMPLE_MIDDLE	Input data sampled in middle of interval
Data_SAMPLE_END	Input data sampled at end of interval

Parameter *clk_idl* determines idle state for clock; can have the following values:

Value	Meaning
CLK_Idle_HIGH	Clock idle HIGH
CLK_Idle_LOW	Clock idle LOW

Parameter *lth_htl* determines transmit edge for data. It can have the following values:

Value	Meaning
LOW_2_HIGH	Data transmit on low to high edge
HIGH_2_LOW	Data transmit on high to low edge

Note

In order to keep this working, you shouldn't override the settings made by the procedures spi_init or spi_init_ordinary as it uses some of the PIC MCU resources.

Pins RC3, RC4, RC5 are configured as needed (don't change TRISC settings for these pins - procedure will set them automatically).

Read and Write

The following routines are provided for comfortable use of master mode :

```
procedure SPI_write(Data: byte);
```

Write byte b to SSPBUF; immediately starts the transmission.

```
function  SPI_read(Buffer: byte)
```

Provide clock by sending data (byte b) and read the received data at the end of the period.

Example

The folowing code demonstrates how to use SPI library procedures and functions. Same example along with m7219.ppas file is given in folder ../mikroPascal/examples. Assumed HW configuration is: max7219 (chip select pin) is connected to RC1, and SDO, SDI, SCK pins are connected to corresponding pins of max7219.

```
program SPI;

uses m7219;

var i : byte;

begin
      SPI_init;             // standard configuration
      TRISC:=TRISC and $Fd;
      max7219_init;         // initialize max7219
      ClearBit(PORTC,1);    // select max7219
        SPI_write(1);       // send address (1) to max7219
        SPI_write(7);       // send data (7) to max7219
      SetBit(PORTC,1);      // deselect max7219
    end;
end.
```

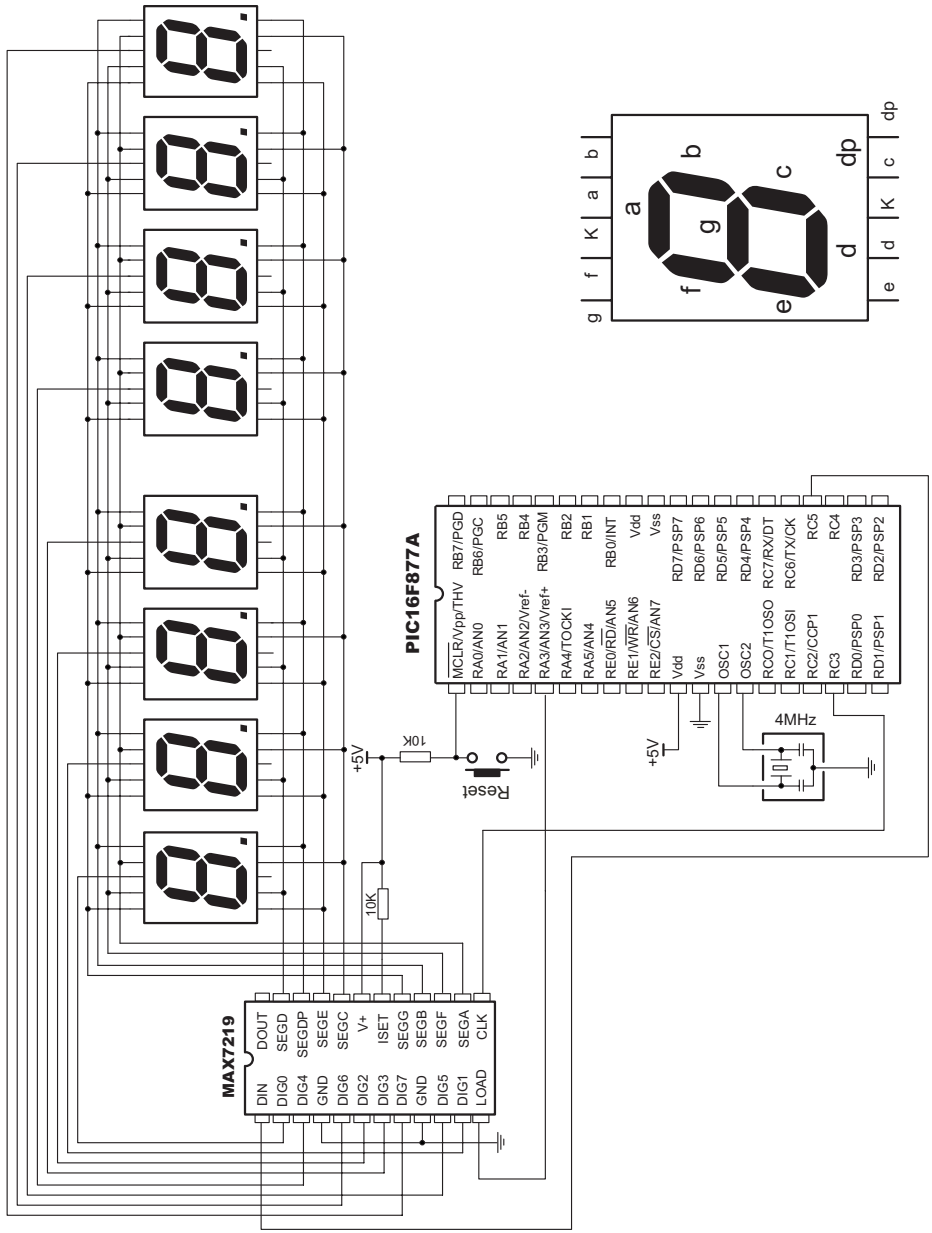


Figure: Example of interfacing MAX7219 with PIC MCU via SPI

USART Library

USART (Universal Synchronous Asynchronous Receiver Transmitter) hardware module is available with a number of PIC MCU models. Set of library procedures and functions is listed below to provide comfortable work with the Asynchronous (full duplex) mode.

You can easily communicate with other devices via RS232 protocol (for example with PC, see the figure at the end of this chapter - RS232 HW connection). You need a PIC MCU with hardware integrated USART (for example, PIC16F877). Then, simply use the functions and procedures described below.

Note

Note that these functions and procedures support module on PORTB, PORTC or PORTG, and won't work with modules on other ports. Examples for PIC MCUs with module on other ports can be found in your mikroPascal installation folder, subfolder 'examples'.

Routines

```
procedure USART_Init(const Baud_Rate: LongInt);
procedure USART_Write(Data: byte);
function  USART_Data_Ready: byte;
function  USART_Read: byte;
```

Certain PIC MCU models with two USART modules, such as P18F8520, require you to specify the module you want to use. Simply append the number 1 or 2 to procedure or function name - for example, USART_Write2(Data : byte);

```
procedure USART_Init(const Baud_Rate: longint);
```

Parameter *Baud_rate* is the desired baud rate;

Example:

```
USART_init(2400);
```

This will initialize PIC MCU USART hardware and establish the communication at baud rate of 2400.

Refer to the device data sheet for baud rates allowed for specific Fosc. If you specify the unsupported baud rate, compiler will report an error.

In order to keep this working, you should not override settings made by the procedure USART_init as it uses some of the PIC MCU resources. (For example: pins RC7, RC6 configured as input, output respectively; do not change TRISC settings for this pins - procedure will set them automatically). Check the figure on the following page.

Following routines can be used after the communication has been established:

```
function  USART_Data_Ready: byte;
```

Return 1 if data is ready; return 0 if there is no data..

```
function  USART_Read: byte;
```

Receive a byte; if byte is not received return 0.

```
procedure USART_Write(Data: byte);
```

Transmit a byte.

Example

The following code demonstrates how to use USART library procedures and functions. When PIC MCU receives data via rs232 it immediately sends the same data back. If PIC MCU is connected to the PC (see figure below), you can test it using mikroPascal terminal for RS232 communication, menu choice Tools > Terminal.

```
program RS232com;

var  Received_byte : byte;

begin
  USART_init(2400);                    // initialize USART module
  while true do
  begin
    if USART_data_ready = 1 then       // if data is received
    begin
      Received_byte := USART_read;     // read received data
      USART_write(Received_byte);      // send data via USART
    end;
 end;
end.
```

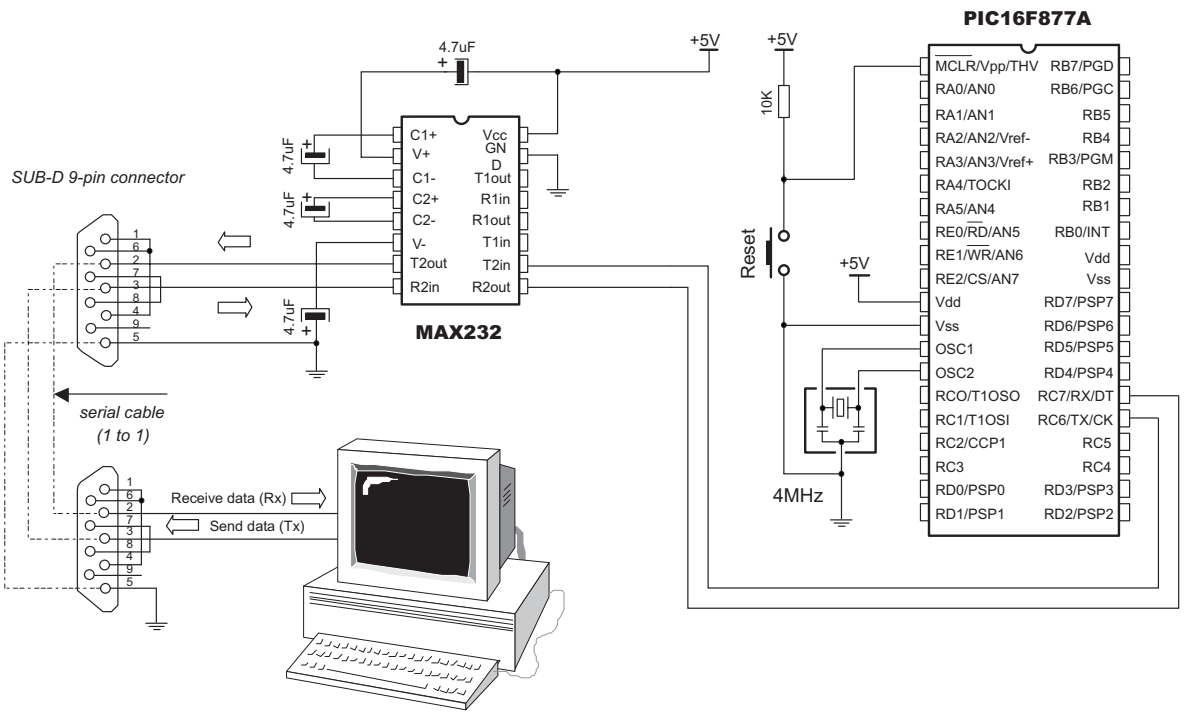


Figure: RS232 HW connection

Software I2C

mikroPascal provides routines which implement software I2C. These routines are hardware independent and can be used with any MCU. Software I2C enables you to use MCU as Master in I2C communication. Multi-master mode is not supported. Note that these functions and procedures implement time-based activities, so the interrupts must be disabled when using them.

I2C interface is serial interface used for communicating with peripheral or other microcontroller devices. Routines below are intended for PIC MCUs with MSSP module. By using these, you can configure and use PIC MCU as master in I2C communication.

Routines

```
procedure Soft_I2C_Config(var Port : byte; const SDA, SCL : byte);
```

Parameter <Port> specifies port of MCU on which SDA and SCL pins will be located; parameters <SCL> and <SDA> need to be in range 0..7 and cannot point at the same pin.

```
procedure Soft_I2C_Start;
```

Issues START condition.

```
function Soft_I2C_Write(Data : byte) : byte;
```

After you have issued a start or repeated start you can send data byte via I2C bus; this function also returns 0 if there are no errors.

```
function Soft_I2C_Read(Ack : byte) : byte;
```

Receive 1 byte from the slave; and sends not acknowledge signal if parameter Ack is 0 in all other cases it sends acknowledge.

```
procedure Soft_I2C_Stop : byte;
```

Issues STOP condition.

Example

This code demonstrates use of software I2C routines. PIC MCU is connected (SCL,SDA pins) to 24c02 EEPROM. Program sends data to EEPROM (data is written at address 2). Then, we read data via I2C from EEPROM and send its value to PORTC, to check if the cycle was successful.

```
program soft_I2C_test;

var  EE_adr, EE_data : byte;
     jj : word;

begin
    Soft_I2C_config(PORTD,3,4);  // initialize full master mode
    TRISC := 0;                  // designate portc as output
    PORTC := $ff;                // initialize portc
    Soft_I2C_Start;              // I2C start signal
    Soft_I2C_Write($a2);         // send byte via I2C
    EE_adr := 2;
    Soft_I2C_Write(EE_adr);      // send byte(address for EEPROM)
    EE_data := $aa;
    Soft_I2C_Write(EE_data);     // send data(data to be written)
    Soft_I2C_Stop;               // I2C stop signal

    for jj := 0 to 65500 do      // pause while EEPROM writes data
      nop;

    Soft_I2C_Start;              // issue I2C start signal
    Soft_I2C_Write($a2);         // send byte via I2C
    EE_adr := 2;
    Soft_I2C_Write(EE_adr);      // send byte(address for EEPROM)
    Soft_I2C_Start;              // I2C signal repeated start
    Soft_I2C_Write($a3);         // send byte (request data)
    EE_data := Soft_I2C_Read(0); // read the data
    Soft_I2C_Stop;               // issue I2C_stop signal
    PORTC := EE_data;            // show data on PORTD
noend:                           // endless loop
    goto noend;
end.
```

Software SPI

mikroPascal provides routines which implement software SPI. These routines are hardware independent and can be used with any MCU.

Note

Note that these functions and procedures implement time-based activities, so the interrupts need to be disabled when using them.

Routines

```
procedure Soft_SPI_Config(var Port : byte;
                          const SDI, SD0, SCK : byte);
procedure Soft_SPI_Init(var Port : byte);
procedure Soft_SPI_Write(Data : byte);
function  Soft_SPI_Read(Buffer : byte) : byte;
```

Configure and initialize SPI using the procedure `Soft_SPI_Config`. Example:

```
Soft_SPI_Config(PORTB,1,2,3);
```

This will set SPI to master mode, clock = 50kHz, data sampled at the middle of interval, clock idle state low and data transmitted at low to high edge. SDI pin is RB1, SDO pin is RB2 and SCK pin is RB3.

Parameter <Port> specifies port of MCU on which SDI,SDO and SCK pins will be located; parameters <SDI>, <SDO> and <SCK> need to be in range 0..7 and cannot point at the same pin;

In order to keep this working, you shouldn't override the settings made by the procedures soft_spi_config as it uses some of the PIC MCU resources. Specified pins SDI,SDO and SCK are configured as needed (don't change TRISX settings for these pins - procedure will set them automatically).

The following functions are provided for comfortable use of master mode:

```
procedure Soft_SPI_Write(Data : byte);
```

Immediately transmit byte Data.

```
function Soft_SPI_Read(Buffer : byte) : byte;
```

Provide clock by sending data (byte Buffer) and return the received data.

Example

This code demonstrates how to use Software SPI procedures and functions. Assumed HW configuration is: max7219 (chip select pin) is connected to RD1, and SDO, SDI, SCK pins are connected to corresponding pins of max7219.

```
program Soft_SPI_test;

include 'm7219.ppas';

var i : byte;

begin
     Soft_SPI_Config(portd,4,5,3);      // standard configuration
     TRISC := TRISC and $Fd;
     max7219_init;                      // initialize max7219
     SetBit(PORTD, 1);                  // select max7219
     Soft_SPI_Write(1);                 // send address to max7219
     Soft_SPI_Write(7);                 // send data to max7219
     ClearBit(PORTD, 1);                // deselect max7219s
end.
```

Software UART

mikroPascal provides routines which implement software UART. These routines are hardware independent and can be used with any MCU. You can easily communicate with other devices via RS232 protocol. Simply use the functions and procedures described below.

Note

Note that these functions and procedures implement time-based activities, so the interrupts need to be disabled when using them.

Routines

```
procedure Soft_UART_Init(var Port : byte;
                         const RX, TX, Baud_Rate : byte);
function  Soft_UART_Read(var Msg_received : byte) : byte;
procedure Soft_UART_Write(Data : byte);
```

Parameter <Port> specifies port of MCU on which RX and TX pins are located (RX and TX have to be on the same port, obviously); parameters <RX> and <TX> need to be in range 0..7 and cannot point the same pin; <Baud_Rate> is the desired baud rate.

Example:

```
Soft_UART_Init(portb, 1, 2, 9600);
```

This will initialize software UART and establish the communication at baud rate of 9600. Maximum baud rate depends on PIC MCU clock and working conditions.

In order to keep this working, you should not override settings made by the procedure `Soft_UART_Init` as it uses some of PIC resources. (the example above configures pins RB1 and as input; do not change TRISB settings for these pins - procedure will set them automatically).

Following functions can be used after the communication has been established:

```
function Soft_UART_Read(var Msg_received : byte) : byte;
```

Function returns a received byte; parameter <Msg_received> will take true if transfer was succesful. `Soft_UART_Read` is a non-blocking function call, so you should test <Msg_received> manually (check the example below).

```
procedure Soft_UART_Write(Data : byte);
```

Procedure transmits a byte.

Example

This code demonstrates how to use software UART procedures and functions. When PIC MCU receives data via RS232 it immediately sends the same data back. If PIC MCU is connected to the PC, you can test it using the mikroBasic terminal for RS232 communication, menu choice Tools > Terminal.

Be aware that during transmission, software UART is incapable of receiving data - data transfer protocol must be set in such a way to prevent loss of information.

```
program soft_UART_test;

var Received_byte : byte;
    Rec_ok : byte;

begin
  Soft_UART_Init(PORTB,1,2,2400);       // initialize software UART
  while true do
  begin
    repeat
      // read received data
      Received_byte = Soft_UART_Read(Rec_ok);
    until Rec_ok;
       Soft_UART_Write(Received_byte); // send data via UART
  end;
end.
```

Flash Memory Library

This library provides routines for accessing microcontroller Flash memory. Note that routines differ for PIC16 and PIC18 families.

Routines

For PIC18:

```
procedure Flash_Write(Address:longint; var Data:array[64] of byte);
function Flash_Read(Address : longint) : byte;
```

For PIC16:

```
procedure Flash_Write(Address : word; Data : word);
function Flash_Read(Address : word) : word;
```

Procedure `FlashWrite` writes chunk of data to Flash memory (for PIC18, data needs to exactly 64 bytes in size).

Procedure `FlashRead` reads data from the specified <*Address*>.

Important

Keep in mind that this function erases target memory before writing <*Data*> to it. This means that if write was unsuccessful, your previous data will be lost.

Example Demonstration of Flash Memory Library for PIC18:

```
program flash_pic18;

const FLASH_ERROR   : byte = $FF;
      FLASH_OK      : byte = $AA;

var toRead, i : byte;
    toWrite : array[64] of byte;

begin
  TRISB := 0;                           // PORTB is output

  for i := 0 to 63 do                   // initialize array
    toWrite[i] := i;

  // write contents of the array to the address 0x0D00
  Flash_Write($0D00, toWrite);

  // verify write
  PORTB  := 0;                          // turn off PORTB
  toRead := FLASH_ERROR;                // initialize error state

  for i := 0 to 63 do
    begin

    // read 64 consecutive locations starting from 0x0D00
      toRead := Flash_Read($0D00 + i);

      if toRead <> toWrite[i] then      // stop on first error
        begin
          PORTB := FLASH_ERROR;         // indicate error
          break;                        //    and stop verify
        end

      else

        begin
          PORTB := FLASH_OK;            // indicate no error
        end;
    end;
end.
```

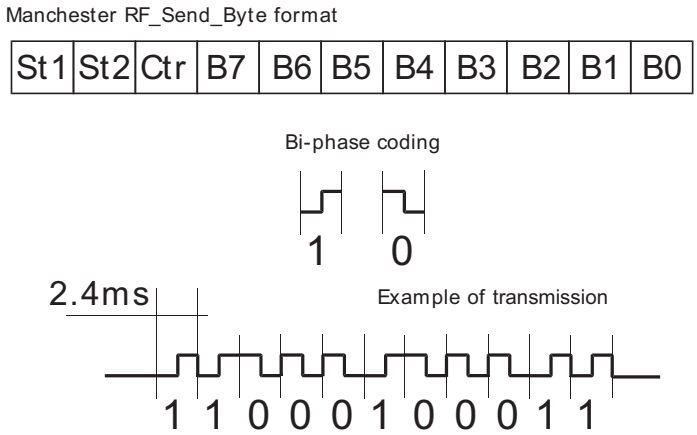
Demonstration of Flash Memory Library for PIC16:

```
program flash_pic16;

const FLASH_ERROR : byte = $FF;
      FLASH_OK    : byte = $AA;

var toRead : word;
    i : word;

begin
  TRISB := 0;                           // PORTB is output

  for i := 0 to 63 do                   // initialize array
    begin
      // write the value of i starting from the address 0x0A00
      Flash_Write(i + $0A00, i)
    end;

  // verify write
  PORTB  := 0;                          // turn off PORTB
  toRead := FLASH_ERROR;                // initialize error state

  for i := 0 to 63 do
    begin

      // read 64 consecutive locations starting from 0x0A00
      toRead := Flash_Read($0A00 + i);

      if toRead <> i then               // stop on first error
        begin

          // i contains the address of the erroneous location
          i := i + $0A00;

          PORTB := FLASH_ERROR;         // indicate error
          break;                        //    and stop verify
        end

      else

        begin
          PORTB := FLASH_OK;            // indicate no error
        end;
    end;
end.
```

Manchester Code Library

mikroBasic provides a set of library procedures and functions for handling Manchester coded signal.

Manchester code is a code in which data and clock signals are combined to form a single self-synchronizing data stream; each encoded bit contains a transition at the midpoint of a bit period, the direction of transition determines whether the bit is a "0" or a "1"; second half is the true bit value and the first half is the complement of the true bit value (as shown in the figure below).



Note

Manchester receive routines are blocking calls (`Man_Receive_Config, Man_Receive_Init, Man_Receive`). This means that PIC will wait until the task is performed (e.g. byte is received, synchronization achieved, etc).

Routines for receiving are limited to a baud rate scope from 340 ~ 560 bps.

Routines

```
procedure Man_Receive_Config(var port : byte; rxpin : byte);
procedure Man_Receive_Init(var port : byte);
function  Man_Receive(var error : byte) : byte;
procedure Man_Send_Config(var port : byte; txpin : byte);
procedure Man_Send_Init(var port : byte);
procedure Man_Send(data : byte);
```

```
procedure Man_Receive_Config(var port : byte; rxpin : byte);
```

This procedure needs to be called in order to receive signal by procedure `Man_Receive`. You need to specify the port and rxpin of input signal. In case of multiple errors on reception, you should call `Man_Receive_Init` once again to enable synchronization.

```
procedure Man_Receive_Init(var port : byte);
```

Procedure works same as `Man_Receive_Config`, with default pin setting (pin 6).

```
function  Man_Receive(var error : byte) : byte;
```

Function extracts one byte from signal. If format does not match the expected, <*error*> flag will be set True.

```
procedure Man_Send_Config(var port : byte; txpin : byte);
```

Procedure needs to be called in order to send signals via procedure `Man_Send`. Procedure specifies <*port*> and <*txpin*> for outgoing signal (const baud rate).

```
procedure Man_Send_Init(var port : byte);
```

Procedure works same as Man_Send_Config, but with default pin setting (pin 0).

```
procedure Man_Send(data : byte);
```

This procedure sends one <*data*> byte.

Example Following code receives message in Manchester code:

```
program RRX;

var error,
    ErrorCount,
    IdleCount,
    temp : byte;
    LetterCount : byte;

begin
  ErrorCount := 0;
  TRISC      := 0;              // Error indicator
  PORTC      := 0;
  Man_Receive_Config(PORTD,6);  // Synchronize receiver
  LCD_Init(PortB);              // Initialize LCD on PORTB
  while true do                 // Endless loop
    begin
      IdleCount := 0;           // Reset idle counter
      repeat
        begin
          temp := Man_Receive(error);  // Attempt byte receive
          if error then
            inc(ErrorCount)
          else
            PORTC := 0;
          if ErrorCount > 20 then      // If too many errors
            begin                      //  try to syncronize again
              ErrorCount := 0;
              PORTC := $AA;            // Indicate error
              Man_Receive_Init(PORTD); // Synchronize receiver
            end;
         inc(IdleCount);

        // If nothing is received after some time
        //   try to synchronize again

          if IdleCount > 18 then

          begin
            IdleCount: = 0;
            Man_Receive_Init(PORTD);   // Synchronize receiver
          end;
        end;
      until temp = $0B;                // End of message marker


  { continues.. }
```

```
{ ..continued }

    // If no errors, then write message

      if error = false then
        begin
          LCD_Cmd(LCD_CLEAR);
          LetterCount := 0;
          while LetterCount < 17 do    // Message is 16 chars long
            begin
              inc(LetterCount);
              temp := Man_Receive(error);
              if error = false then
                begin
                  LCD_Chr_CP(temp);
                end
              else
                begin
                  inc(ErrorCount);
                  break;
                end;
            end;
          temp := Man_Receive(error);
          if temp <> $0E then
            begin
              inc(ErrorCount);
            end;
        end;
    end;
end.
```

Example Following code sends message in Manchester code:

```
program RF_TX;

var i: byte;
    s1: string[20];

begin
  PORTB  := 0;                        // Initialize port
  TRISB  := %00001110;
  ClearBit(INTCON, GIE);              // Disable interrupts
  Man_Send_Config(PORTB,0);           // Initialize manchester sender
  while TRUE do
    begin
      Man_Send($0B);                  // Send start marker
      Delay_ms(100);                  // Wait for a while
      s1 := 'mikroElektronika';
      for i := 1 to Length(s1) do
        begin
         Man_Send(s1[i]);             // Send char
         Delay_ms(90);
        end;
      Man_Send($0E);                  // Send end marker
      Delay_ms(1000);
    end;
end.
```

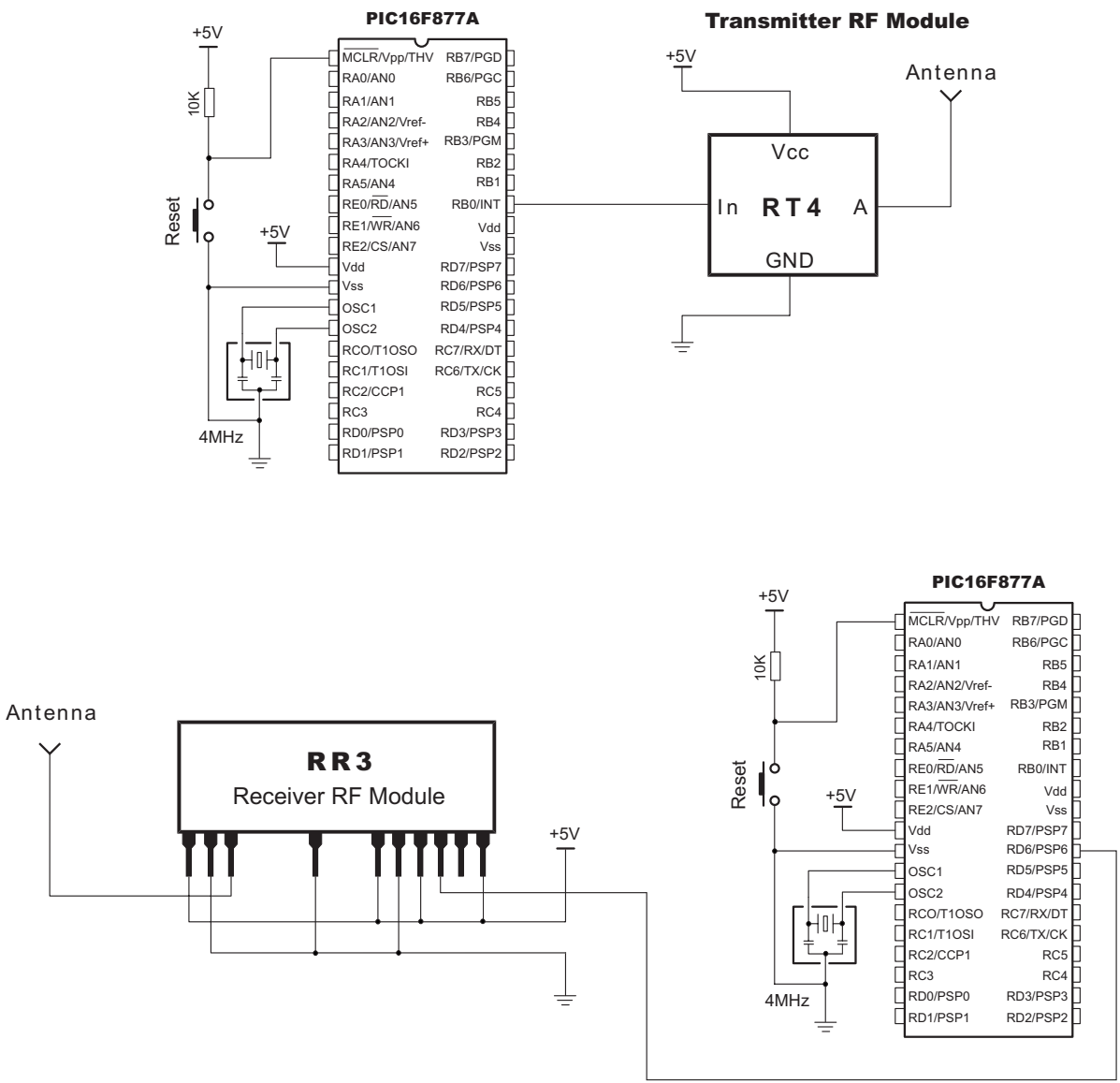


Figure: simple Transmitter and Receiver connection

Numeric Formatting Routines

Numeric formatting routines convert byte, short, word, and integer to string, and can also convert decimal values to BCD and vice versa.

Routines

You can get text representation of numerical value by passing it to one of the routines listed below:

```
procedure ByteToStr(input: byte;  var txt: array[6] of char);
procedure WordToStr(input: word;  var txt: array[6] of char);
procedure ShortToStr(input: short; var txt: array[6] of char);
procedure IntToStr(input: integer; var txt: array[6] of char);
```

Parameter *input* represents numerical value of that should be converted to string; parameter *txt* is passed by address and it contains the result of conversion. All four procedures behave in similar fashion for appropriate input data type. (Parameter *txt* has to be of sufficient size to fit the converted string.)

Following routines convert decimal values to BCD (Binary Coded Decimal) and vice versa:

```
function Bcd2Dec(bcd_num : byte) : byte;
function Dec2Bcd(dec_num : byte) : byte;
function Bcd2Dec16(bcd_num : word) : word;
function Dec2Bcd16(dec_num : word) : word;
```

For instance, function Bcd2Dec converts 8-bit BCD numeral *bcd_num* to its decimal equivalent and returns the result as byte. Simple example:

```
..
var a, b: byte;
begin
  a := 140;
  b := Bcd2Dec(a);    // b equals 224 now
end.
```

Example

The following code demonstrates use of library procedure ShortToStr. Example prints the converted value to LCD display.

```
program num_test;

var txt : array[20] of char;
    i : short;
begin
  PORTB := 0;                    // initial value for PORTB
  TRISB := 0;                    // PORTB is output
  LCD_Init(PORTB);               // initialize LCD on PORTB
  LCD_Cmd(LCD_CLEAR);            // send command 'clear display'
  LCD_Cmd(LCD_CURSOR_OFF);       // send command 'cursor off'

  // print txt, 1st row, 1st col
  LCD_Out(1,1,'mikroElektronika');
  Delay_ms(1000);

  txt := 'testing..';             // write string to txt
  LCD_Out(2,1,'testing..');
  Delay_ms(1000);
                                  // print txt, 2nd row, 1st col
  LCD_Cmd(LCD_CLEAR);
  for i := 127 downto -111 do
  begin
     ShortToStr(i,txt);           // convert variable i to string
     LCD_Out(2,1,txt);            // print i (string value)
     Delay_ms(100);
     LCD_Cmd(LCD_CLEAR);
  end;
   txt := 'The End';
   LCD_Out(1,1,txt);
end.
```

Trigonometry Library

Trigonometric functions take an angle (in degrees) as parameter of type word and return sine and cosine multiplied by 1000 and rounded up (as integer).

Routines

Functions implemented in the library are:

```
function sinE3(Angle : word) : integer;
function cosE3(Angle : word) : integer;
```

Functions take a word-type number which represents angle in degrees and return the sine of <*Angle*> as integer, multiplied by 1000 (1E3) and rounded up to nearest integer:

result = round_up(sin(Angle)*1000)

Thus, the range of the return values for these functions is from -1000 to 1000.

For example:

```
var angle : word;
   result : integer;

angle  := 45;
result := sinE3(angle);    // result is 707
```

Note

Parameter <Angle> cannot be negative.

These functions are implemented as lookup tables. The maximum error obtained is ±1.

The example in the following page demonstrates use of library functions sinE3 and cosE3. Example prints the deg, sine and cosine angle values on LCD display. The angle parameter can be altered by pushbuttons on PORTC.0 and PORTC.1.

Example

```
program TestTrigon;

var angle : word;
    txtNum : array[6] of char;
    res : integer;
    chFlag : byte;

begin
  TRISB := 0;
  TRISC := $FF;
  LCD_init(PORTB);
  LCD_Cmd(LCD_CURSOR_OFF);
  angle := 45;
  LCD_Out(1,1,'deg');
  LCD_Out(1,6,'sin');
  LCD_Out(1,12,'cos');

  chFlag := 1;

  while True do
  begin
    if (Button(PORTC, 0, 1, 1)=True) and (angle < 1000) then
    begin
      inc(angle);
      chFlag := 1;
    end;
    if (Button(PORTC, 1, 1, 1)=True) and (angle > 0) then
    begin
      dec(angle);
      chFlag := 1;
    end;

    if chFlag=1 then
    begin
      chFlag := 0;
      LCD_Out(2,1,'                ');
      WordToStr(angle,txtNum);
      LCD_Out(2,1,txtNum);
      res := sinE3(angle);
      IntToStr(res, txtNum);
      LCD_Out(2,6,txtNum);
      res := cosE3(angle);
      IntToStr(res, txtNum);
      LCD_Out(2,12,txtNum);
      Delay_ms(100);
    end;
  end;
end.
```

Sound Library

mikroPascal provides a sound library which allows you to use sound signalization in your applications.

Routines

```
procedure Sound_Init(var Port: byte; Pin: byte);
procedure Sound_Play(Period: byte; Num_Of_Periods: word);
```

Procedure `Sound_Init` initializes sound engine at specified <*Port*> and <*Pin*>. Parameter <*Pin*> needs to be within range 0..7.

Procedure `Sound_Play` plays the sound at the specified port pin. <*Period_div_10*> is a sound period given in MCU cycles divided by ten, and generated sound lasts for a specified number of periods (<*Num_of_Periods*>).

For example, if you want to play sound of 1KHz: T = 1/f = 1ms = 1000 cycles @ 4MHz. This gives us our first parameter: 1000/10 = 100. We'll play 150 periods like this:

```
Sound_Play(100, 150)
```

Example

This is a simple demonstration of how to use sound library for playing tones on a piezo speaker. The code can be used with any MCU that has PORTB and ADC on PORTA. Sound frequencies in this example are generated by reading the value from ADC and using the lower byte of the result as base for T (f = 1/T).

```
program SoundADC;

var adcValue: byte;

begin
  PORTB  := 0;                    // Clear PORTB
  TRISB  := 0;                    // PORTB is output
  INTCON := 0;                    // Disable all interrupts
  ADCON1 := $82;                  // Configure VDD as Vref,
                                  //    and analog channels
  TRISA  := $FF;                  // PORTA is input

  Sound_Init(PORTB,2);            // Initialize sound on PORTB.RB2
  while true do
    begin
      adcValue := ADC_Read(2);    // Get lower byte from ADC
      Sound_Play(adcValue, 200); // Play the sound
    end;
end.
```

Utilities

mikroPascal provides a set of procedures and functions for faster development of your applications.

Routines

```
function Button(var PORT: byte; PIN, Time, Astate: byte): byte;
```

The Button function eliminates the influence of contact flickering due to the pressing of a button (debouncing).

Parameters *PORT* and *PIN* specify the location of the button; parameter *Time* represents the minimum time interval that pin must be in active state in order to return one; parameter *Astate* can be only zero or one, and it specifies if button is active on logical zero or logical one.

Example

This code demonstrates use of library function Button. Example reads the state on PORTB, pin 0, to which the button is connected. On transition from logical 1 to logical 0 which corresponds to release of a button, value on PORTD is inverted.

```
program test;

var oldstate : byte;

begin
  PORTD := 255;
  TRISD := 0;
  TRISB :=  255;
  while true do
  begin
    if Button(PORTB, 0, 1, 1) then oldstate:= 255;
    if  oldstate and  Button(PORTB, 0, 1, 0) then
       begin
         portD := 0;
         oldstate := 0;
       end;
  end;
end.
```

Contact us:

If you are experiencing problems with any of our products or you just want additional information, please let us know.

Technical Support for compiler

If you are experiencing any trouble with mikroPascal, please do not hesitate to contact us - it is in our mutual interest to solve these issues.

Discount for schools and universities

MikroElektronika offers a special discount for educational institutions. If you would like to purchase mikroPascal for purely educational purposes, please contact us.

Problems with transport or delivery

If you want to report a delay in delivery or any other problem concerning distribution of our products, please use the link given below.

Would you like to become mikroElektronika's distributor?

We in mikroElektronika are looking forward to new partnerships. If you would like to help us by becoming distributor of our products, please let us know.

Other

If you have any other question, comment or a business proposal, please contact us:

MikroElektronika magazine
Admirala Geprata 1B
11000 Belgrade
EUROPE

Phone: + 381 (11) 30 66 377, + 381 (11) 30 66 378
Fax: + 381 (11) 30 66 379
E-mail: *office@mikroelektronika.co.yu*
Web: *www.mikroelektronika.co.yu*